NOTE N-878

SOFTWARE TECHNOLOGY TRANSFER AND EXPORT CONTROL

Seymour E. Goodman, University of Virginia, Chairman
Norman S. Glick, Department of Defense
William K. McHenry, University of Virginia
John B. McLean, Rome Air Development Center
Claude E. Walston, IBM, Federal Systems Division
Clark Weissman, System Development Corporation

January 1981

Prepared for
Office of the Under Secretary of Defense for Research and Engineering

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION
400 Army-Navy Drive, Arlington, Virginia 22202

Contract MDA 903 79 C 0018
Task T-0-072

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A106 809	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Software Technology Transfer and Export Control		5. TYPE OF REPORT & PERIOD COVERED December 1979-October 1980
		6. PERFORMING ORG. REPORT NUMBER IDA Note N-878
7. AUTHOR(s) Seymour E. Goodman, Chairman, Norman S. Glick, William K. McHenry, John B. McLean, Claude E. Walston, Clark Weissman		8. CONTRACT OR GRANT NUMBER(s) MDA 903 79 C 0018
9. PERFORMING ORGANIZATION NAME AND ADDRESS INSTITUTE FOR DEFENSE ANALYSES 400 Army-Navy Drive Arlington, VA 22202		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task T-0-072
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy Under Secretary for International Programs and Technology, OUSDR&E The Pentagon, Washington, DC 20301		12. REPORT DATE January 1981
		13. NUMBER OF PAGES 143
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) DoD/IDA Management Office 400 Army-Navy Drive Arlington, VA 22202		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE None

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

None

18. SUPPLEMENTARY NOTES

N/A

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software, Export Control, Technology Transfer, Technology Transfer Mechanisms, Militarily Critical Technologies List, Software Life Cycle, Software Development Tools, Software Engineering

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The formulation of a reasonable and effective export control policy for software products and software engineering know-how has been an important and difficult task for both the U.S. Government and industry. This note represents the contribution of the Software Subgroup of Technical Working Group 7 (Computers) to the development of a technical policy for the control of software.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20.

The note attempts to provide useful discussions and analyses in three areas. The first examines why the problem of software control has been so difficult, and presents the rationale for a number of working hypotheses which underlie the approach taken for the entire study. The second examines, in some detail, the "what" (know-how and operational capability) and "how" (transfer mechanisms) of software technology transfer. The last section recommends several items for inclusion on the Militarily Critical Technologies List.

FOREWORD

This note reports the recommendations and views of the Software Subgroup of Technical Working Group 7 (Computers) of the Critical Technologies Project. The work reported herein provided part of the basis for Chapter 6 of the report of Technical Working Group 7. However, some of the views expressed herein differ from those presented in the output of the Critical Technologies Project as a whole, in that the rationale of this note interweaves considerations of control policy with determination of which technologies should be identified as Militarily Critical Technologies. The Critical Technologies Project dealt only with the latter issue in the FY 1981 study effort. Nevertheless, since this note represents a thoughtful contribution to the understanding of the relationship between software technology and its control for export purposes, it is being published as part of the related work performed in support of the Critical Technologies Project.

This note has been reviewed by IDA management, but the views expressed herein are those solely of the authors and should not be construed to be those of IDA nor its DoD sponsor.

ABSTRACT

The formulation of a reasonable and effective export control policy for software products and software engineering know-how has been an important and difficult task for both the U.S. Government and industry. This note represents the contribution of the Software Subgroup of Technical Working Group 7 (Computers) to the development of a technical policy for the control of software.

The note attempts to provide useful discussions and analyses in three areas. The first examines why the problem of software control has been so difficult, and presents the rationale for a number of working hypotheses which underlie the approach taken for the entire study. The second examines, in some detail, the "what" (know-how and operational capability) and "how" (transfer mechanisms) of software technology transfer. The last section recommends several items for inclusion on the Militarily Critical Technologies List.

TABLE OF CONTENTS

Executive Summary

<u>Recommended List Entries</u>

The software subgroup of the Computers Technical Working Group (TWG-7) recommends the addition of eight software technologies to the Militarily Critical Technologies List (MCTL). These are:

1. Software Life-cycle Management Technology (Section 8.2.1)
2. Software Library Data Base (Section 8.2.2)
3. Software Development Tools (Section 8.2.3)
4. Maintenance of Large Software Products (Section 8.2.4)
5. Formal Methods and Tools for Developing Trusted Software (Section 8.2.5)
6. Secure Software (Section 8.2.6)
7. Large Self-Adapting Software Systems (Section 8.2.7)
8. Commercial Software Integral to Critical Military Systems (Section 8.2.8)

The concern of the software subgroup has been with <u>software technology</u> for the development of large software systems. It can be argued that the relative U.S. and NATO superiority over the U.S.S.R. and the Warsaw Pact in developing and maintaining large, integrated software and software-hardware systems is one of the most critical advantages remaining to the West in military-related technologies.

A large software system is one whose effective development and maintenance require sophisticated life-cycle management technology. We try to capture the essence of this technology in the first five recommended list items. Our posture is to view software more as a process than a product, i.e., as a labor intensive social activity of teams of technical personnel working over a long life cycle to produce a variety of "products" such as English and mathematical documentation of the requirements, functional specifications, user operations, and source and object code - what we collectively call "software." Our recommendations focus on this process which is necessary to construct large software systems that have sufficient functionality to provide significant military utility. When software size reaches a threshold it requires many people, automated tools, various types and levels of documentation, and technical and management controls employed over a multi-year development and use life cycle. It is this technology and technical know-how which we recommend for control, and we believe that such control is best imposed through the control of selected transfer mechanisms and software development tools. See Section 2 for a more extensive discussion.

In terms of somewhat simplistic parameters, we have chosen the rough equivalent of 15,000 lines of source code (without normal comment statements and documentation) or four person-years of effort for initial system development as threshold values for defining "large." Except for small programs of direct and important military value, which should probably be controlled through classification, we believe that the government should not be concerned with the enormous volume of software that falls beneath our threshold values. The burden on both the government and the software community would be counterproductive, and the security-threatening technology transferred via such products is not likely to be significant. A more complete set of "capture-release" criteria are presented in Section 3.3.

The remaining three list items are concerned with more specialized technologies. Two cover fairly general emerging technologies of high military utility. Other technologies in this class may have to be added later, since constraints of time and expertise did not permit us to thoroughly cover all software areas. One such future possibility is robotics software. The final item concerns know-how that could enable adversaries to counter or jam U.S. military capabilities by acquiring access to and compromising software used in critical military systems.

Applications Software

It must be emphasized that we are not concerned with the operational capabilities provided by applications software. This is beyond the scope of our expertise. For example, from our perspective, a CAD/CAM package for designing aircraft engines, shipped in object code form with only user manuals for documentation, transfers little software know-how. It may well transfer a great deal of technology for designing aircraft engines. We assume that applications software of concern will be flagged by TWGs or DoD groups dealing with the applications technologies.

Listed below are a few of the conclusions, other than MCTL items, that have emerged from this study. See also Sections 3 and 8.3.

Major Implementation Issues

Software is a much more open technology than hardware. Care must be taken to avoid methods of restricting technology transfer which would significantly and adversely affect the advancement of this technology in the U.S. One of the great strengthes of U.S. software technology is its extraordinary openness. In spite of intensive "borrowing" of software by other countries, we continue to make more extensive use of product and know-how transfers within the U.S. than is possible from the U.S. to foreign countries. In fact, with respect to our NATO allies, we believe that joint efforts should be expedited by the U.S. government. The best way to stay ahead in the race is to run faster.

The software subgroup rejects the idea that virtually all software should not be controlled because "they can get it anyway" through covert means. Forcing an adversary to resort to covert means to obtain software products adds links to the acquisition chain which may increase the acquisition time, risk, effort, and the probability that information will be lost during the transfer process. Such information losses will play a more prominent role in hindering the acquisition of large, sophisticated software systems. Furthermore, the lack of ready access in those situations to the the services and other forms of support provided by the organizations which develop the software can further degrade the usefulness of what is acquired.

From an implementation standpoint, the best ways to avoid the two undesirable extremes mentioned in the preceding two paragraphs are through:

1. An informed software community,

and 2. The careful analysis and control of software transfer mechanisms.

The software community consists of a large number of people and organizations in government, academia, and industry. Much of this community has not considered its activities to fall under export control prior review or even self restraint. The control of software will involve a more delicate and difficult balance of interests between academic freedom, free enterprise, and national security than has been the case for any other technology. The software community needs to be involved in determining this balance. To a large extent, the effectiveness of any set of controls will be dependent on the voluntary restraint of an informed professional community.

We believe that a lot of software can be sold to adversary countries, and that a lot of person-to-person contacts between software specialists from the U.S. and adversary countries is possible without significantly threatening U.S. national security. The key to drawing a line between what is and what is not desirable and controllable is an understanding of the effectiveness and controllability of the software technology transfer mechanisms. We have made a fairly comprehensive analysis of these mechanisms (Section 6), and our findings are summarized in Section 8.1 of this report. We believe the <u>recommendations for controlled transfer mechanisms</u> that we have included with each MCTL item in Sections 8.2.1-8.2.8 <u>should be considered part of the definition of the item</u>.

Trends and Leakage

Most current technical trends will make it easier for adversaries to acquire software capabilities through transfer from the West. A thoughtful set of controls may slow this acquisition rate, perhaps significantly, but it must be recognized that there will always be substantial leakage since what is being sought is becoming increasingly widespread and available.

Taxonomic Approaches

We believe that a taxonomic approach to software export control, i.e., what is essentially done now with hardware, is unmanageable for systems and applications software. Furthermore, such an approach misses some of the most important aspects of control of this technology. However, we have used a partial taxonomic approach in our recommendations concerning software development tools. See Section 5, especially Section 5.6.

Product Form and Portability

It is necessary to control the portability and modifiability of software products through technical means. One general and moderately effective way to do this is limiting shipment to object code or hardware forms (e.g., ROM) and basic user manuals. Simple, cheap, effective means to tie a software product to a specific configuration are needed.

In general, the transfer of any software product should be uncontrolled if the product does not provide a direct military operational capability or explicitly fall into a small number of categories (e.g., software development tools or self-adapting systems), provided that it is transferred in the form of object code with only users manuals and passive maintenance service, and provided that there be reasonable technical safeguards that the system not be portable.

Thought Experiments

Our analysis of transfer mechanisms used a set of novel 'gedanken' (thought) experiments showing the value of what-how transfers to a determined antagonist. One experiment considered how an adversary might piece together a tactical command, control, and communications system from commercially available software. Another investigated what transfer mechanisms were important in the construction of an actual time-sharing system. Both experiments played an important role in formulating our ideas, framework, and recommmendations. See Section 7.

1. Introduction

The formulation of a reasonable and effective export control policy for software products and know-how has been an important and difficult task for both the U.S. Government and industry. This task becomes increasingly important, because the hardware capabilities of our adversaries are improving to the point where it is possible for them to work on a respectable number of fairly large and sophisticated systems of military importance. It can be argued that the relative U.S. superiority over the U.S.S.R. in developing large, integrated software and software-hardware systems is one of the most critical advantages remaining to the U.S. in military-related technologies.

This report represents the contribution of the software subgroup of the Computer Technical Working Group (TWG-7) to the development of a technical policy for the control of software. The starting point for our analysis was the conceptual framework established by the software subgroup of the Computer Networks Critical Technology Expert Group (CNCTEG)[4,7], and our assessment of the software capabilities of our adversaries. We have expanded and modified the CNCTEG framework as necessary on the basis of new material and our own deliberations. Although we have pushed forward both the breadth and depth of the analysis of software technology transfer, the short working period available to the TWG made it impossible to comprehensively study all of software in great detail.

Software is a particularly difficult technology to analyze and control. We try to explain why in Section 2. The problem of software technology transfer and export control will continue to require attention. We think it necessary to provide a discussion of our conceptual basis so that readers can better understand the rationale for our conclusions and recommendations, and so that this report may serve as a building block for further studies.

Section 3 contains a detailed listing of the working hypotheses that have emerged from our conceptual basis, and a brief summary of our perceptions of foreign and adversary capabilities. This section provides the foundation from which our analysis and recommendations follow, and we feel that this foundation is as important as the recommendations themselves.

An overview of the 'what' and 'how' of software technology transfer is contained in Section 4. We define some basic terminology and provide an introduction to our top-down taxonomies. The topmost levels of these breakdowns are also presented here.

Section 5 contains our detailed taxonomy of the 'what' of software technology transfer. This analysis is limited to systems software and software development tools. A partial evaluation of military utility is also provided.

The mechanisms for software technology transfer, i.e., the 'how', are treated in Section 6. We provide a taxonomy, and a discussion of the possible effectiveness and controllability, of the various mechanisms. Detailed analyses are given of one moderately active mechanism (this analysis is contained in the classified appendix to this section), and of one passive mechanism - the physical forms of product sales. These two examples were chosen because of their importance and wide use, and it is hoped that they may also serve as prototypes for further analyses.

Section 7 is built around examples of 'gedanken' experiments that try to pull the 'who', the 'what', and the 'how' together. A software development team based on our perception of a reasonably good Soviet group is hypothesized to want to build two major systems of considerable military value. The systems have been chosen from among those with which members of our TWG software subgroup are deeply familiar. The thought experiments estimate the effect of several technology transfer mechanisms on the ability of our hypothetical Soviet-like development team to produce the target systems. The analyses are done from both the standpoint of the acquisition of a significant operational capability and from the standpoint of the acquisition of in-depth know-how transfer.

Section 8 contains our recommended entries for the Militarily Critical Technologies List, and several other recommendations, concerning problems of implementation.

A variety of software terms used in the body of the report were felt to be part of the working vocabulary of most software specialists. Readers who are unfamiliar with these terms should consult the DACS Glossary [5].

The software subgroup of the Computers Technical Working Group (TWG-7) consisted of: Norman S. Glick (Department of Defense), Seymour E. Goodman (Chairman, University of Virginia), William K. McHenry (University of Virginia), John B. McLean (Rome Air Development Center), Claude E. Walston (IBM,FSD), and Clark

Weissman (System Development Corporation).

We would like to thank the Science Division of the U.S. Army Foreign Science and Technology Center (USAFSTC-SD) and William Carlson and others at the Defense Advanced Research Projects Agency (DARPA) for their assistance and the use of their facilities. We would also like to thank the more than 50 other people from government, industry, and the academic community who gave their time to share their views and experience with us.

As of July 31, 1980, the material presented in this report does not represent an official view of any U.S. government organization, nor does it reflect the views of any other organization. Conversely, any changes made to this report after this date will not necessarily reflect the views of the software subgroup. With the exception of the separated SECRET appendix to Section 6 on transfer mechanisms, no classified material was used in the preparation of this report. The authors welcome constructive comments.

2. Why Is Software Control So Elusive?

A better grasp of software and its control is obtained if we rethink our implicit model of software as a product, like a dozen eggs or a computer of specified performance, and view it as a social process. Software that contains a large number of complex functions is generally developed by a large number of people working in a complex social structure. Furthermore, studies have revealed as much as an order of magnitude difference in the performance of individual programmers [cf. 3]. This phenomenon is also at work in the Soviet Union, since we do not have a lock on bright people, and therefore they have the ability to recreate software produced by single individuals. If we adopt a process-oriented view of software, we can better reveal what we are trying to control and why it is so difficult. Then we can begin to formulate an approach to controlling software technologies and certain goods which should not be exported to adversary countries.

2.1. The Software Life Cycle

The best working software is a product of a number of discrete stages with defined output and review, i.e., a social process. Together with the use of the software library inventory and software development tools, they comprise what we have chosen to call "life-cycle management technology." Although numerous approaches to the life cycle for software are used, one that is modelled here is the DoD methodology.

The earliest stage is that of concept definition, when the overall system purpose and operation is conceived. A clear statement of objectives is required. Objectives may be derived from higher-level systems, from control of lower-level systems, from simulations, and from "war gaming" scenarios. Cost and scheduling factors also assist in the concept definition.

The _requirements and specifications_ stages begin to structure what the system must do to satisfy its objectives. Again, simulation can be employed. Techniques of structured requirements are usefully employed to follow the flow of system operational control and its needed data and computational requirements. Once developed, these requirements and their specifications must be written in well-formed, unambiguous notation. A number of such languages now exist and are used in preparing mathematically precise system specifications of what the system must do, i.e., its service specifications.

System design is the stage that defines how the system works, i.e., how the system implements the service specifications. A design may be written in at least one of a number of notations: flow diagrams and data diagrams, state machine tables or specification languages, English, structured English, or even a programming Higher Order Language (HOL) of the coding variety. Each approach carries with it advantages and disadvantages and a considerable technical methodology. All approaches use a form of modular design which defines the input, output, and functional behavior of each module. With the definition of these modules and their interfaces to other modules, system hardware, and human components, a software architecture is developed. More modern methods go further in describing the types of parameters, and their "visibility" in scope to other modules. Side effects and environment considerations for each module may also be specified.

Coding proceeds directly from the design stage. First the individual modules are coded, then the associated modules until a chain of integrated modules is built up which performs one or more of the service specifications. The design is often implemented in an HOL such as FORTRAN, COBOL, JOVIAL, PASCAL, etc.

These module chains, called "builds," relate directly to the requirements specification and form the basic unit of testing of the system. Modern testing methods employ "threads" or "builds" testing which checks the correct operation of a thread (i.e., a logical, ordered subset of the whole) of functions which satisfy one system requirement. There is considerable technology required for system testing. Test plans must be developed to lay out a strategy of tests to be performed in sequence by a number of systems people working in parallel. Test conditions are set up, parameters to drive modules are created, and results are captured and analyzed against specifications and requirements. Errors that are found must be logged and engineering changes generated and controlled for correcting such errors. The module data base grows and changes with each engineering change, and a system of controls tracks software releases and the errors outstanding against them. These tests proceed thread by thread until all requirements are demonstrated. Threads are then merged into a complete integrated system, which is tested for correctness and performance. Finally, the system is tested at the user's installation. This may be the first time that all of the system elements work together and use real ("live") data.

A number of management techniques are used at various points in the life-cycle. Standard management activities, e.g., preparing work breakdowns, cost estimates, schedules, and manpower loading statements, have been adapted to the peculiarities of the software industry. Some of the know-how which has been acquired through difficult learning experiences includes understanding and anticipating the rigor needed for a large software project, handling detailed internal management and technical documents, incorporating a number of defined events and milestones for management review at various levels, using modelling and control systems, and building project management on the basis of hierarchies of individuals who have different levels of experience and responsibility. Much of this experience cannot be acquired through open, passive sources.

A recent technological development which is changing traditional patterns of testing is the strategy of "building the system correctly," rather than testing for flaws. This technology employs formal mathematical specifications of the system in a predicate calculus language and a set of software tools that operate on the formal specifications to prove their correctness mathematically. By successive refinement of these formal specifications, a hierarchy of increasingly more detailed specifications are written and proved correct. These specifications then become the coding specifications from which HOL code is written and it too is subject to proofs (using tools) as part of the formal verification of the correctness of the software. This is an emerging technology which shows promise of reducing the cost of software development and maintenance, and of improving the quality and correctness of the software.

After final testing, the system enters operation and maintenance (O&M). The O&M stage might be considered the end, as the system is completed and in operational use. But large programs are quite complex and involve many interfacing interrelated "clockwork" mechanisms. They are very fragile in the sense of needing continuing support and maintenance to keep them current and operational. Repairs are needed to correct errors, upgrades to improve performance, changes to accommodate hardware configuration changes or new performance requirements. New functional capabilities unforeseen or unclearly outlined in the original specifications may need to be added. Such modification of working software is the rule of industry and is reflected in the model, version, or release numbers associated with all software products.

Such repair is really redesign and requires a return to earlier life cycle stages. Thus, O&M must make extensive use of the documentation of the program contained in the software library data base (see below). Knowledge required to carry out O&M involves understanding the software architecture, detail design, the various specification and programming languages in which the software is written, the computer on which the software operates, the complement of equipment in the system configuration, the applications environment, the types and ranges of expected input and output variables, how to operate test tools, and the proper use of configuration management tools to keep the software current. O&M is a major problem stage for software, because it is entered years after the concept stage and when few of the original designers are available to perform the changes. It has been found in major military systems that it costs 100 times more to fix a problem in the O&M stage than to detect and repair an error in the requirements and analysis stage. Overall, O&M costs about twice the total of all other stages combined. While O&M follows the other life-cycle phases sequentially, it really must be considered an entirely separate stage. Few, if any, of the personnel who built the software continue to support it and O&M is carried out for an extended period of time after operation begins. We intend to address O&M as a separate candidate technology for the militarily critical technologies list.

That is the life cycle of a typical large scale software system for commercial or military application. Small scale software development mimics these stages in a less structured manner and, therefore, in a less controllable way. A problem for export control is to determine at what stage the control is applied. For some systems, diverse teams of diverse vendors at diverse locations work on different stages, each having different output elements of the partially completed system. We contend that export control may more effectively address the process of software development through these life cycle stages rather than through the software products. Furthermore, the control must address the large scale software developments, not the smaller efforts; since these larger efforts offer greater visibility, controllability, and capabilities of higher value not easily emulated by adversaries.

For our purposes, a large scale software system is one that employs the social processes of meetings, interaction, coordination, cooperation, and documentation of progress that apply for a project with four or more people working for a year or more. Typically, such an effort results in the initial delivery of a system with from 15-20,000 to hundreds of thousands of computer instructions of HOL statements. (A more formal statement of our "capture-release" criteria is contained in Section 3.3.)

2.2. The Elusive Form of Software

Unlike finished goods in other technical manufacturing, software has no single physical form. In the early life cycle stages it is English functional descriptions. In design stages it is more formal mathematics or logical information flow specifications. In the coding stage it is in the form of "source" text in a HOL. This text is translated by software tools into "object" binary form for direct execution on a given computer. Application programs in HOL source code form may be translated into many different object code forms for different computers, or into different object code forms on the same computer for different configurations of interfacing software and peripheral hardware. And for each form there must be accompanying documentation to describe the software operation and differences.

Finally, through the O&M process, the source programs are changed, repaired, and improved in function and performance to produce an assortment of new versions of essentially the same "product." The aggregate of these software items typically constitute millions of lines of text in various forms. If any one of these items is incorrectly formulated or maintained, incorrect operation can result.

In order to maintain these items over the course of the software life cycle, a software library data base is used. The data base is created incrementally and is a "living" document, best maintained on-line by a sophisticated set of tools. The structure of the data base depends on the conventions of the languages and notations employed in the various stages of development. These conventions must permit both human and machine access to the text.

The data base contains multiple directories of the objects in the data base. All directories originate from a master directory, which is often organized along system or component lines so that releases of modules are placed with other modules of the same thread or function. Subdirectories often follow the organizational structure of the development project, with each programmer having his or her private files. These files are periodically released to the software librarian to include in the master directory. The master directory is further organized by text forms for each development stage; it then resembles a multi-dimensional matrix of functions, forms, and people. This structure is key to the retrieval and to the automation of the generation of system products: software and documentation. Even the naming of objects becomes a crucial technology, since much of the structure is embedded in the names, they reflect a path through the directories, and they encode the form of the text and version number of interest (e.g., JONES.PASCAL.3). They also provide a uniform key upon which all of the related software tools can operate.

Access to the library is strictly controlled by the librarian and operating procedures for obvious reasons of safety and protection, but also for less obvious reasons of error control, cost control, status reporting, and project communication. Private files are strictly controlled by each owner. Backup procedures are of highest priority and are handled both by the librarian and individual programmers. Backup becomes more crucial and more costly as the library grows.

All of the techniques outlined above are collectively known as "configuration management." As an important component of life-cycle management technology, configuration management techniques are well-developed in the United States and a critical technology which is not widely available or appreciated in the Soviet Union.

2.3. Software Development Tools

Software development tools play a key role in life-cycle management technology. Although they are products, a large amount of know-how is required to use them in the context of life-cycle management technology. Hence, the software subgroup feels that comprehensive controls on life-cycle management technology should involve controls on tools as well. They include library maintenance tools, composition tools, translation tools, quality control and verification tools, and administration tools. A more complete taxonomy of these tools is presented in Section 5.5.

2.4. Why the Control of a "Product" is Not the Key Issue

Our posture is to recognize software as a process and to impose controls on the production of large scale software, principally through the control of transfer mechanisms (Section 6) and the tools for life cycle software development. With some exceptions, we emphasize the development process, controlling the process of software development itself, rather than products which might reveal critical technologies, because software is ubiquitous, easy to transport, and dependent on O&M. The uses of software have proliferated to the extent that any attempt to catalogue them all would require huge amounts of manpower and resources (see Section 5).

Software is easy to transport both mechanically and electronically. That alone makes it difficult to control. However, modern computer usage further compounds the problem by making computer systems remotely accessible. Therefore, the benefits of the software may be obtained without the need to own a copy. These remote transaction services are a growing sector of the computer service business encouraged by new telecommunications tariffs.

Finally, software is difficult to control since much of the business does not involve software products, but support services and technical assistance, in the form of training, documentation, consulting, and O&M. This technical assistance is critical for the end user; without it he would incur a large cost and lead time if he had to maintain the software himself.

2.5. Software Versus Hardware Development

We hope to highlight the nature of software by the following comparison with hardware. Superficially, software development appears similar to hardware development. Both move through similar phases: concept, specification, design, production, operations, and maintenance. Both employ skilled staffs and complex tools. Both require competent and experienced management, and both are expensive. But since software is a process, the transfer of technology implicit in full software sales goes far beyond what would be considered reasonable for the sale of hardware. The sale of a turnkey plant, for example, does involve a transfer of mass production techniques for the product and spare parts, and the technology needed for operation and maintenance. However, if the adversary wishes to enhance and improve the product, he must master all the necessary engineering skills himself. Full software sales implicitly assume the transfer of the skills, because software never reaches a completed form but is constantly being maintained and enhanced. Other differences between hardware and software are apparent in each part of life-cycle management technology.

1. The software life cycle

Although a number of the stages of development of hardware and software are similar, there are some major differences.

First, the technology which we are trying to control is much more complex than for hardware, since software is among the most complex "machines" man has ever built. Typical products involve millions of statements, lines of text or code. The software objects are "soft" and have to be referenced by naming each uniquely. The naming process to manage the software objects is itself complex. The order and structure of these objects are as important as the objects themselves, and the placement must also be uniquely identified by name or structural content.

The most complex hardware objects built today are the LSI chips for computer memories. The smallest component level is the "bit" flip-flop transistor which is replicated many times. The biggest chips today have approximately 64,000 bits; a number which is at least an order of magnitude smaller than the number of lines of code in medium-large software systems. And the information content of a bit component is far less than the information content of a software statement or line of code. Secondly, the testing requirements for software differ considerably from those for hardware. Both technologies employ prototypes as vehicles to test readiness of the product for market. Upon completion of the hardware testing, the major cost and production engineering still remain to produce the product. Not so with software; much of the cost and effort is consumed in completing the prototype. In fact, the prototype _is_ the first version of the product. Of course the tools and testing methods are radically different in form and substance. With software it is often difficult to specify the test conditions. Furthermore, the tools for testing may have to be built from scratch for each product. There is little reusable test software. And the problems are compounded because the new software test tools must themselves be tested. There is no appreciable technology transfer or "learning curve" from one project to another in software testing.

Finally, the economics of the development process are significantly different. Hardware is machine intensive after the first prototype. Up to that time hardware and software are similar handcrafted items, except that software is orders of magnitude more complex. The major hardware costs come after prototyping with production engineering, tooling, and materials. Software has no counterpart to post prototype hardware production costs. Hardware production decisions can be based on retail and OEM quantity orders based on the prototype's performance and the unit cost (total cost divided by units produced). Software is labor intensive throughout its life cycle; there are no materials, and rarely any quantity production. Indeed, most software products are customized for the end-user system configuration. The nature of the software business drives the manufacturer to write off his total cost on a few units, often just one. The government's cost-plus contracting is a typical example of this form of software economics. Retail, high volume, software sales have not been a staple of the software industry, and existing cases are often tied to hardware sales.

2. Software Library Management

We have repeatedly underscored the fact that there is not really an end product in software. Software must be considered the sum total of the contents of a software library data base at any given moment in time. While hardware products are visible, tangible items, software products must be input into a machine before they can be "seen." It is impossible to view the entirety of a software product at once. Hardware has its documentation, but for software, the documentation is the product because any current code form is just a documentation of the development process so far.

3. Tools

Both technologies employ tools and some computer tools are common to both. But hardware needs radically different tools for materials handling, shipping, attaching, measuring, etc. Software tools involve invented languages, mathematics, symbol manipulation and management, and a set of procedures and software programs for processing the languages. In hardware production retooling is a major engineering, management, and economic consideration which comes long after a working model or prototype exists. And the tooling is designed for the production of a discreete number of products over a certain useful life.

Software tools are often effectively built from scratch for each job with an impact akin to hardware retooling; and the tools never wear out. Recent software engineering thrusts have been to attempt to standardize on some of these tools. The difficulty is that each new product has different constraints. Software tools built for one product cannot easily address the requirements of a different CPU, set of peripherals, programming language, or even different development methods which must be used in building another product. That selection of constraints is made long before the software issues are addressed. It would be the height of folly for a hardware product customer to tell the manufacturer what he wants, how it is to work, how it is to be built and tested, and that the factory must be built on his premises in some end user location which is hundreds of miles away from the manufacturer's facility. Yet that is exactly the situation for the majority of DoD and other government software procurements.

3. Working Hypotheses

Listed below are the working hypotheses that have emerged from the deliberations of the TWG-7 software subgroup. These hypotheses underlie many of the analyses and recommendations of this study.

3.1. Scope of Our Analysis

a) The primary concern of the software subgroup has been with software technology for the development of large software systems. We are not concerned with the operational capabilities provided by applications software. This is beyond the scope of our expertise. For example, from our perspective, a CAD/CAM package for designing aircraft engines, shipped in object code with only users manuals for documentation, transfers little software know-how. It may well transfer a great deal of technology for designing aircraft engines. We assume that applications software of concern will be identified by Technical Working Groups concerned with applications technologies.

b) Software technology differs in several important ways from most other military-related technologies (see Section 2). These differences complicate the problem of export control. As a result of these differences and complications, we do not feel constrained to analyze software within the context of existing rigid formats, such as the Commodity Control List (CCL). Such a product-oriented, taxonomic approach has been tested in Section 5 and found deficient (see Section 5.6).

c) It is not within the purview of the software subgroup to pass judgment on the relative social merits of "academic freedom," "free enterprise," and "national security." We are aware of, and in some cases sympathetic to, arguments as to whether or not various forms of academic or commercial exchanges have political or social value that compensates for concomitant undersirable technology transfers. We leave such decisions to others. It is our task to make technical evaluations of various know-how and product transfers, and it is inconsistent to be concerned about a particular transfer mechanism when it is used by a commercial organization, but to ignore the use of the same mechanism by a nonprofit organization. Therefore, our analyses and recommendations are made without regard for originating organization or price.

3.2. Limitations of Software Export Control

a) Most current trends will make it easier for adversaries to acquire software capabilities, through transfer from the West. A thoughtful set of controls may slow this acquisition rate, but it must be recognized that there will always be substantial leakage since what is being sought is becoming increasingly widespread and available.

b) It will be possible to find counterexamples or to conjecture counterexample scenarios to almost every broad generalization. For example, specific counterexamples could be found to any attempt to rank the effectiveness of What and How transfer combinations, although the ranking would seem to hold for most situations. This is the result of the enormous diversity of possible software transfer situations.

c) Several of the countries against whom export controls are directed have considerable scientific and technological resources and accomplishments. It is unreasonable to assume that they are incapable of making significant advances in software development on their own. Nevertheless, they seem to rely on the transfer of software technology from the U.S. and other COCOM countries. We do not understand to what extent this reliance is a matter of convenience and to what extent it is a matter of dependence. It may be possible to slow the progress of software development in adversary countries with a thoughtful set of controls. However, to try and provide a quantitative measure of what effect any given set of controls would have on further progress in these countries is unrealistic.

3.3. Software That Should Be Considered for Export Control

3.3.1. The Extremes of Software Export Controls

a) Care must be taken to avoid methods of restricting technology transfer which would significantly and adversely affect the advancement of software technology in the U.S. One of the great strengths of U.S. software technology is its very openness. In spite of extensive "borrowing" of software by the other countries, we continue to make more extensive use of product and know-how transfers within the U.S. than is possible from the U.S. to the foreign countries. The best way to stay ahead in the race is to run faster. It may be of use to drop some obstacles behind us to slow our opponents or to avoid helping them. But we should not slow down to do this, particularly since our adversaries have demonstrated some ability to get around obstacles. Fortunately, our opponents have handicapped themselves in software development in ways that are at least as effective as anything we could do.

b) The software subgroup rejects the idea that virtually all software should not be controlled because "they can get it anyway" through covert means. Forcing an adversary to resort to covert means to obtain software products adds links to the acquisition chain which may increase the acquisition time, risk, effort; and the probability that information will be lost during the transfer process. Such information losses will play a more prominent role in hindering the acquisition of large, sophisticated software systems. Furthermore, the lack of ready access in those situations to the the services and other forms of support provided by the organizations which develop the software can further degrade the usefulness of what is acquired.

3.3.2. Capture-release Criteria

a) Size and complexity

Software that is large enough and complex enough to require some sort of social process and team effort to build and maintain is most likely to be of concern. As rough guidelines, we recommend the equivalent of at least 15,000 lines of sources code (plus normal comment-type documentation above the source code line count) or at least four person years of effort devoted to the development of the product before it is initially delivered to the user. Furthermore, the software should represent an integrated system, not some simple collection of easily programmed pieces. We explicitly reject minimum monetary cost as a "threshold variable" because there is no widely accepted method for computing such costs.

b) Destination Country and End User

At one extreme, it goes without saying that we should control the sale of militarily critical software to military or police force of an adversary country. At the other extreme (and we feel this should be equally obvious) sales of useful software to the COCOM countries and, in particular, to their military establishments, should be encouraged and expedited. Unfortunately, this has not always been the case. In between there is a spectrum running from our close NATO allies to our potential adversaries. For example, given the great differences between the capabilities of the People's Republic of China and the Warsaw Pact countries, for example, it is impossible to treat the two "equally" without some special understanding of the use of that term. Such decisions must ultimately be made from a political, rather than technical, standpoint.

c) Product Form

Product form should play a large role as a criterion for export controls. Products which are very portable can be used in a large number of installations. The military establishment in the U.S.S.R., for example, enjoys extraordinary appropriation privileges over the resources of the general economy, increasing the risk that products which are portable will be vulnerable to military exploitation. Products in source code form can be adapted to other uses, modified, or enhanced with greater ease than products in object code. A further discussion of the differences between object and source code is in Section 6.4.5., and recommendations for technical measures to make products less portable are considered in Section 6.5.2.

d) Operational Capability

From our perspective, the transfer of any software product should be uncontrolled if it does not provide a direct militarily critical operational capability as outlined in the reports of the other TWGs, or if it does not fall into four other categories we isolate in Section 8; provided that it is transferred in the form of object code with users manuals and passive maintenance service, and that there are reasonable **technical** safeguards that the system not be portable.

e) Volume of Sales

We have not been able to come up with a meaningful criterion for export control which involves volume of sales, i.e., the number of copies of a product sold.

f) Arrays of Software Products

The software subgroup has found the problem of arrays of software products to be particularly intractable. We recognize, on the one hand, that arrays of products can offer an operational capability which is much greater than the operational capability provided by the products alone. On the other hand, it is difficult to prevent an adversary from amssing such arrays on a piece-by-piece basis. Shipping products in a form which does not make them very adaptable, e.g., object code, can make it more difficult to integrate stand-alone products. Also, we are capturing many products that are already integrated. Section 7.3.1 addresses this problem in greater detail. The software subgroup recommends that more research be done in this area.

3.4. Adversary Capabilities and Foreign Availability

Brief summaries of the software subgroup's perceptions of adversary and foreign software capabilities are presented below. Given the enormous diversity of software technology and the large number of programmers in the world, these summaries could not be anything but crude oversimplifications. Within each country, it is likely that there exist individuals or groups whose talents and experience greatly surpass our perceptions of the norm for each nation.

3.4.1. Adversary Capabilities

a) The political-economic structure of the communist countries in general, and of the Soviet Union in particular, is such that the military, state police, and related organizations enjoy what, by Western standards, are extraordinary appropriation privileges with respect to the resources of the general economies. Furthermore, the Soviet Union appears to have extraordinary privileges over and access to the scientific and technical resources available in most of the Warsaw Pact countries and elsewhere (e.g., Cuba). This is not to say that the Soviet military or KGB can always take effective advantage of these privileges, but the opportunity to do so is well established. Most of the software technology that is useful for building civilian systems is also applicable to developing military systems, and it is often transferable without diversion of equipment. These considerations make it exceptionally difficult to separate military from civilian use of software products and technology, and to separate consideration of the U.S.S.R. from the other CEMA countries.

b) The U.S.S.R. has, by far, the greatest software capability of any of the communist countries. Nevertheless, much of this capability is rudimentary by U.S. standards. An extensive discussion of non-military Soviet software developments and problems, through 1978, is given in [8]. Progress since then has been evolutionary, although the volume of Soviet software work has increased fairly dramatically in recent years. This increase has been the result of the greater availability of respectable hardware and of greater experience and awareness of the importance of software. Most of the Soviet software commmunity continues to suffer from assorted major and minor systemic difficulties. One of their most critical weaknesses has been in the development and maintenance of large, sophisticated, highly integrated systems involving interfaces with communications and sensor devices.

There is evidence that the Soviets are concerned about this major deficiency, and are trying to do something about it. They have gained experience during the last decade, and have been considerably influenced by U.S. developments. The software subgroup believes that software technology in the U.S.S.R. may have now developed far enough in both quality and quantity for the Soviets to be on the threshold of investing significant resources into what is generally called "software engineering" in the U.S. Their efforts to do this could be aided substantially by certain kinds of product and know-how transfers from the COCOM countries. We address some of these transfers in Sections 5-7.

c) Several of the East European Warsaw Pact countries (notably the German Democratic Republic, Hungary, Czechoslovakia, and Poland) have software capabilities that, on a per capita basis, are as strong as those of the U.S.S.R. The company GDR Robotron, in particular, seems to have capabilities in the systems software area that are comparable to those of some respectable West European companies. Some of these countries are known to be working on software systems that could have significant military value. Bulgaria and Romania seem to be somewhat weaker in software technology.

d) The available evidence [c.f. 2,6] leaves little doubt that the People's Republic of China has a rather rudimentary national software capacity, although there seem to be some very knowledgeable individual Chinese and, as in virtually everything else, the potential of that country is enormous. China is not now in a position to develop many large, sophisticated, militarily critical software systems, and this is likely to be the case for at least the next few years. The U.S. and the other COCOM countries need to give serious thought to what sort of aid they want to provide China in this technology.

e) At least two other adversary countries, Cuba and North Korea, have software capabilities, although these are much weaker than those of other countries we have considered. We know almost nothing about North Korea. Cuba is a participant in the SM CEMA minicomputer effort and organizations concerned with national-level software development have been identified.

f) The software subgroup has not been able to identify clearly any integrated systems of software development tools in use in adversary countries, although it appears that some partially integrated sets are in use. In recent years, there has been a substantial increase in the volume of literature describing tools, but most of these tools are stand-alone and are concerned with composition and translation. Increased efforts to acquire this technology have followed a growing awareness of its importance.

3.4.2. Foreign Availability

a) The U.S. leads the world in software technology. However, the U.S. lead is by no means a monopoly, although it may still be very substantial in some important areas. Many of the COCOM countries, and several non-COCOM countries as well, have substantial software development capabilities. Adversary countries could learn much from non-U.S. sources, and these sources might also serve as a funnel of U.S. software technology to our adversaries. Although tighter unilateral control of software technology by the U.S. might be of some value, the effectiveness of these controls will be limited without the cooperation of COCOM. Some thought might also be given to how to make it more difficult for U.S. software technology to be transferred via non-COCOM third countries.

b) Western Europe and Japan have aggressively sought to acquire software development knowledge. U.S. experts with appropriate software experience have been sought to be consultants, to present papers at foreign conferences, and to teach courses. Western European software engineering expertise, particularly academic, is given widespread recognition. Possessors of strong academic backgrounds in software development methodology are given responsibility in projects to develop large commercial software systems. Despite the strong interest in acquiring software development skills, one finds in Western Europe the same phenomenon that has been experienced in the U.S.: success in large software projects is often dependent on [illegible] experience of having been a part of an earlier large product development effort. The formal knowledge which has been eagerly sought seemingly must be supplemented by the actual experience of building a software system--a frequently painful first-time experience in which the project does not meet all its objectives.

In Japanese industries, as in Western European industries which are controlled or strongly influenced by government, there is a tendency to have a longer planning cycle, encouraging the use of methods whose payoff is more long-range than U.S. companies are willing to accept. This willingness to defer payoff has led in some instances (e.g., robotics) to the development of indigenous software capabilities based on research primarily conducted by U.S. laboratories, even before U.S. companies have been willing to make the investment.

c) Several development groups in Western Europe (COCOM and non-COCOM) and Japan are aware of the importance of software development tools technology. However, the availability of integrated systems of tools and the base of experience in their use (especially in the development and O&M of large military systems) are below that of the United States.

d) Yugoslavia and Finland have limited software capabilities and somewhat delicate political and economic relationships with the Warsaw Pact countries.

4. An Overview of the "What" and "How" of Software Technology Transfer

The CNCTEG software subgroup established what effectively amounted to a top-down, hierarchical framework for the breakdown of software technology transfer. The software subgroup of TWG-7 has chosen to use this framework, with some additions, such as the tree-like representations shown in Figures 4.1, 4.2, and 4.3.

The top level of this breakdown, shown in Figure 4.1, partitions software technology transfer into:

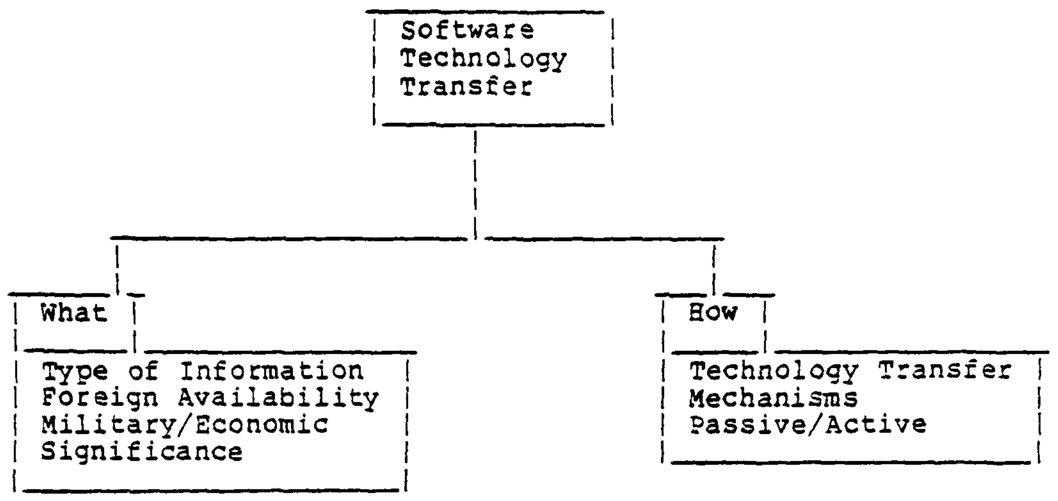
(1) "What" - software know-how and operational capability, where:

(a) know-how is the knowledge to produce and deploy software products, and

(b) operational capability is the capability provided by a product or service to accomplish some task.

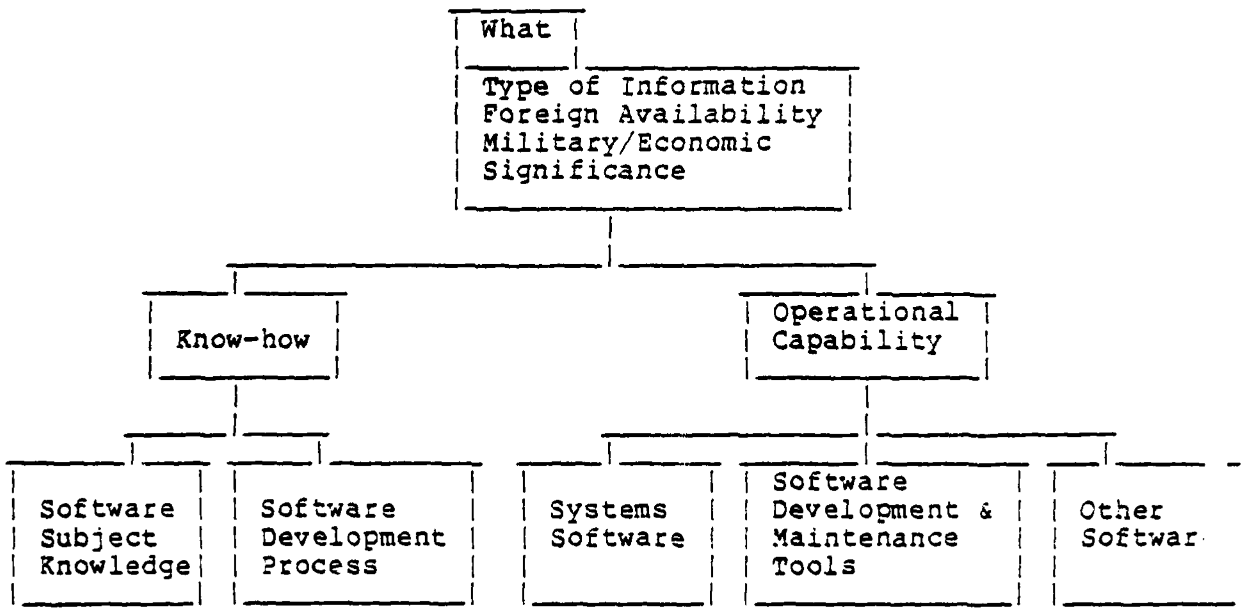
(2) "How" - the mechanisms for the transfer of software know-how and operational capability.

A further breakdown of the "what" of software technology transfer is shown in Figure 4.2. Software know-how is divided into subject matter knowledge (e.g., algorithms, modules, data structures, etc.) and the software development process. Some examples of the basic components which comprise subject matter knowledge are real-time control algorithms, table look-up algorithms, sensor-based algorithms, etc. The software development process includes such stages as concept definition, requirements and analysis, design, coding, testing, operation, and maintenance (see Section 2.1).



Software Technology Transfer
What
Type of Information
Foreign Availability
Military/Economic Significance
How
Technology Transfer Mechanisms
Passive/Active

Figure 4.1. Software Technology Transfer



What
Type of Information
Foreign Availability
Military/Economic Significance
Know-how
Operational Capability
Software Subject Knowledge
Software Development Process
Systems Software
Software Development & Maintenance Tools
Other Softwar

Figure 4.2. The "What" of Technology Transfer

Operational capability is broken down into three categories. The first is systems software, which refers to sets of independent programs which form a functional whole, usually put together in a layered approach (e.g., the combination of microcode, standard machine instructions, and an operating system). Software development and maintenance aids are the tools which are deemed critical to the software development process. Other software applications refers to the capabilities deemed critical to development and manufacturing processes other than software (e.g., CAD/CAM) and capabilities that are directly related to important military activities (e.g., nuclear weapons design).

Section 5 presents a taxonomy of systems software and development and maintenance tools, along with a partial military utility evaluation. A taxonomy and analysis of technology transfer mechanisms (the "how" of technology transfer) comprise Section 6. The top level of this taxonomy is shown in Figure 4.3. Transfer mechanisms can be broken down into three categories. The first is product shipment, which deals with user manuals, programs, etc., delivered to adversary countries. The second is assistance, a broad category which encompasses help or services provided to adversaries. Finally, the remote access usage category refers to the usage of a product by an adversary while the product remains under Western control, e.g., through a computer network.

Section 7 is an attempt to pull together the "what", "how", and "who" of software technology transfer in the form of a few detailed hypothetical scenarios. We felt that such

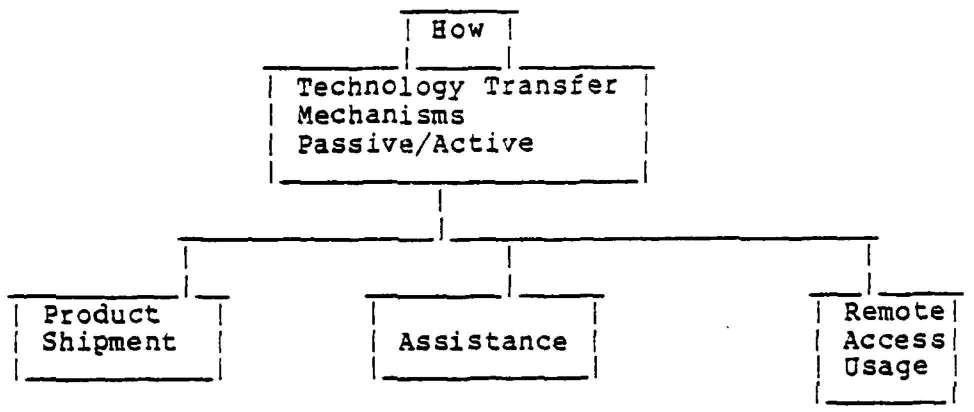


Figure 4.3. The "How" of Technology Transfer

experiments helped to deepen our appreciation of the problems of software technology transfer and its control.

5. Software Know-how and Operational Capability

5.1. Introduction

This section provides a partial taxonomy of software know-how and operational capability which expands on the structure given in Section 4. It does not cover all of software; due to assorted time and manpower constraints, it deals only with systems software and software development tools in detail. An initial effort is made to assess the military utility of many of the listed items.

The software subgroup believes that a taxonomic/military utility approach to software export control is too coarse and unmanageable. However, it is difficult to fully appreciate the problems until one tries to consider such an approach in detail. A discussion of this view is reserved for Section 5.6.

5.2. Know-how

Software know-how has been partitioned into two categories: software subject matter knowledge, and know-how embodied in the software development process.

Software subject matter knowledge consists of the basic "components" of software: algorithms, data structures, etc. Since most of this information is widely available in the open literature, it is beyond control. Any attempt to classify this knowledge would be fruitless due to the sheer size of the task. Furthermore, much of this knowledge is implicitly present in our taxonomy. Application-specific components or programs (CAD/CAM routines, sensor-based algorithms, etc.) which have high military utility are covered separately in the reports of the other TWGs.

Software development process know-how covers the knowledge needed to produce and deploy a broad spectrum of large software products. This know-how is more developed in the United States than in any other country and is based on extensive first-time experiences which often involve building less than completely successful systems. This experience involves the management of a team of software engineers and programmers who are involved with the life-cycle technology discussed in Section 2. For typical defense systems, the product of this effort (i.e., the software library data base which documents the transformation of the product from initial requirement statements to the latest working object code version) can amount to millions of lines of text and code which must be managed.

The military utility of life-cycle management technology is very high. Any large defense system which the adversaries might wish to build requires a knowledge of this technology. Furthermore, the experience gained in building a seemingly innocuous system (e.g., a medical sensor system which handles multiple real-time inputs) can be applied in an entirely different context. Since adversary experience with this technology is somewhat limited, its criticality is also quite high. Therefore, the control of the export of life-cycle management technology is considered of prime importance by the software subgroup.

5.3. Operational Capability

Operational capability is divided into systems software, software development tools, and other software. Our taxonomy addresses the first two categories.

The evaluation of military utility for software was carried out using the matrix framework shown in Figure 5.1. Each category of software was given a rating from

zero: little or no discernible military utility, to

nine: extremely significant military utility.

The category as a whole was then given a rating greater than or equal to the highest rating across the spectrum of possible military applications[12].

The software subgroup cautions that this approach is somewhat arbitrary, only partially and tentatively complete, and needs further refinement. Several problems arise in trying to determine military utility in this fashion. First, one can posit situations in which most software could have a high utility in a given military system. Hence, the ratings for broad classes are simply too general. Secondly, we do not think that an analysis in this form addresses the difficult problem of arrays of software products. This analysis was carried out considering products in isolation. The value of an array of products, such as an integrated set of software development tools as opposed to stand-alone tools, can be much greater. Therefore, we consider these ratings to be useful to the extent that they provide a general ordering of relative utilities of products when used in isolation in specific military systems.

	Military Systems (see Legend* below)						
	I	II	III	IV	V	VI	VII
5.4. Systems Software**	9	3	9	9	9	9	9
4.1. Human Interface Software	8	2	2	2	7	8	7
4.2. Data Capture	8	3	8	1	3	5	4
4.3. Data Presentation	9	1	3	7	8	8	5
4.4. Data Transmission	7	3	5	5	3	7	5
4.5. Data Processing	9	3	9	9	9	9	8
4.6. Data Storage	9	3	5	6	8	8	5
4.7. Data Access	9	3	5	6	8	8	5
5.5. Software Development Tools	9	6	9	9	9	9	9
5.1. Software Library Data Base	9	6	9	9	9	9	9
5.2. Software Support Library Management	9	3	9	9	9	9	9
5.3. Text Processing	3	1	2	1	2	3	1
5.4. Translation	8	2	7	7	5	5	7
5.5. Quality Assurance and Control	9	6	9	9	9	9	9
5.6. Accounting/ Administration	9	6	9	9	9	9	9

Figure 5.1. Matrix of Military Utility of Computer Software Technoligies

Legend

I. Operational Capabilities (Military)
II. Military Business Programs
III. Satellite Systems
IV. Ballistic Missle Systems
V. Strategic C-Cubed and Planning
VI. Tactical C-Cubed
VII. Aircraft Weapon Systems

----

* Ratings for other categories, such as Space Operations, Cruise Missiles, Communications Systems, Shipbased Systems, Submarine Systems, Tank Systems, and Field Artillery have not been supplied.

** Numbers, such as 5.4.3, correspond to subsections that follow.

5.4. Systems Software

Systems software refers to that body of software integrated into a functional whole, or system, serving a specific, common purpose. Often systems software is built from more general-purpose components, but specific to the computer hardware and application area. Current systems software methods advocate a structured or layered architecture of software. At the lowest layer is the hardware. The lowest software layers manage this hardware in the form of routines to control input/output, interrupts, memory allocation, and even the microcoded instruction definitions. Higher layers define increasingly more abstract computer "machines" which present abstract services to yet higher layers, and implement that functionality by combinations of service requests on the lower layers. The collection of layers of software for managing the computer is called a monitor or an operating system (OS). Modern OSs "virtualize" computer resources into a standard, abstract set of resources viewed commonly by all application programs. Still higher layers include software for networks of computers, data base management systems (DBMS), and particular applications.

To provide a taxonomic view of this plexus of interdependent, applications dependent software, we postulate the "generic information system," consisting of seven broad abstract functions provided by all systems in some measure. These functions follow directly the flow of data and its processing in the generic components of a system. Within each function, we list the variety of software processing characteristics of that function. The reader is cautioned that no single system possesses all these components, but is created by integrating some from each function. These large, complex systems are difficult and expensive to construct, and represent the end product of a technical and managerial superiority by the U.S. in life-cycle management technology. Figure 5.2. presents a tree-like overview of systems software.

TAXONOMY	MILITARY UTILITY
5.4.1. Human Interface Software	{8}
1.1. System Operation	{4}
1.2. System Command Language	{3}
1.3. System User Authentication	{8}

Discussion

This area deals with the computer software that permits humans to control system operations and to authenticate that a particular human is authorized to have access to the system. The high rating for system user authentication software reflects the importance of this software for tactical C-cubed and strategic C-cubed military functions. This software is critical in military systems for preventing unauthorized users from obtaining data from the system, obtaining services from the operation of the system, destroying the system, entering incorrect or misleading data, or entering illegal commands to subsidiary military organizations. Knowledge of the details of this class of software could improve adversary chances to identify vulnerabilities in the assocated military systems.

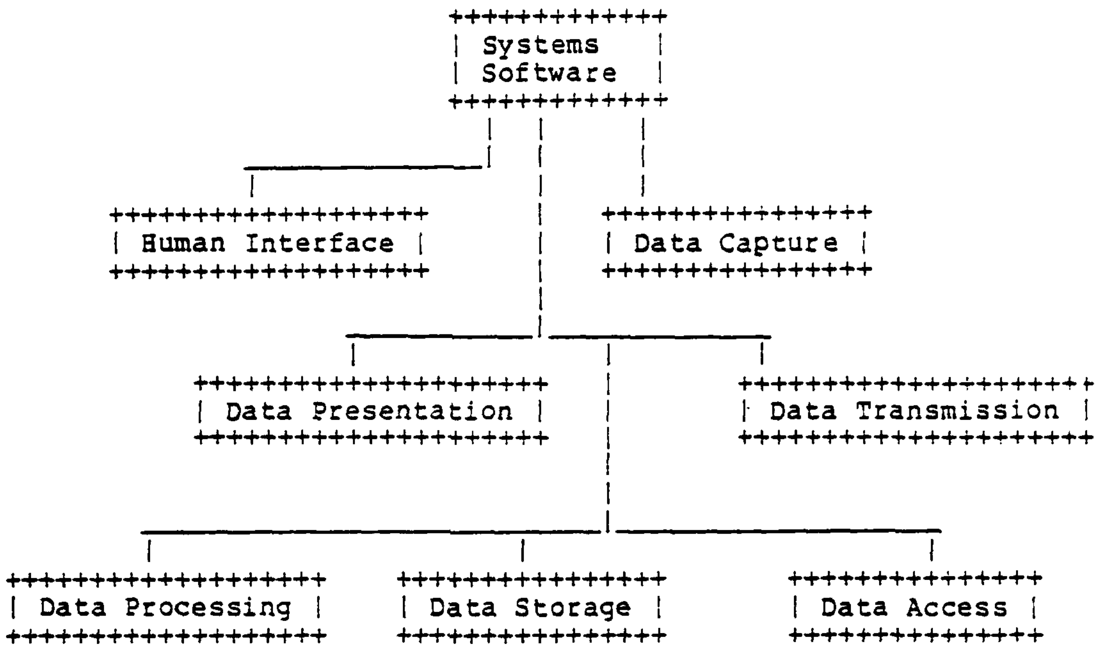


Figure 5.2. Systems Software Taxonomy

TAXONOMY	MILITARY UTILITY
5.4.2. Data Capture Software	{8}
2.1. Sensor Data Input	{8}
1.1. Electromagnetic Sources	{8}
1.1. Satellite Visual	{8}
1.2. Radar	{6}
1.3. Infrared Detection System	{7}
1.4. Other	{?}
1.2. Acoustic Sources	{?}
1.3. Other Sources	{?}
2.2. Human Data Input	{7}
2.1. Natural English Language Input	{7}
2.2. Keyboard/Console Drivers	{2}
2.3. Speech Input Analyses	{7}
2.4. Menu Reader	{4}
2.5. Graphic Inputs	{7}
5.1. Light Pen	{5}
5.2. Track Ball	{3}
5.3. Joy Stick	{3}
5.4. X-Y Tablet	{3}
5.5. Touch Panel	{7}
2.3. Peripheral Input	{5}
3.1. Card Reader	{0}
3.2. Paper Tape	{0}
3.3. Magnetic Tape/Cards	{3}
3.4. Optical Character Reader	{5}
3.5. Bar Code Reader	{3}
3.6. Photo Reader	{5}

Discussion

The high rating {8} given data input software is based on the complexity of the software needed for the satellite input problem. Here the data rate is so high that even the fastest computing engines require intricate real-time software to accommodate the input data.

5.4.3. Data Presentation Software	{9}
3.1. Direct Presentation Software	{?}
1.1. Keyboard Console Drivers	{?}
1.1. Scroll Terminals	{1}
1.2. CRT Terminals	{5}
1.3. Text Formatters	{2}
1.2. Speech Synthesis	{5}
1.3. Menu Presentation	{6}
1.4. Graphic Output	{?}
4.1. CRT	{8}
4.2. Plotters	{8}
4.3. Photo (Film) Camera	{7}
4.4. Thermal Printers	{3}
4.5. Cursor Positioning	{4}
3.2. Output to Peripherals	{?}
2.1. Line Printer	{0}
2.2. Spooling	{0}
2.3. Card Punch	{0}
2.4. Paper Tape	{0}
2.5. Magnetic Tape/Cards	{3}

TAXONOMY	MILITARY UTILITY
5.4.3 (con't)	
2.6. Barcode Printing	{3}
2.7. Phototypesetter	{4}

Discussion

The military utility rating {9} for the data presentation category is based on the importance of this software to strategic and tactical C-cubed systems, each of which has a very high military importance. The rating {8} for software related to graphic CRT and plotter display is based on their use in the presentation of target data, geographical data, and vehicle position and characterization.

5.4.4. Data Transmission	{7}
4.1. Network Protocols	{9}
1.1. Link Level	{3}
1.1. HDLC, SDLC, etc.	{3}
1.2. Transparency Controls	{4}
1.3. Error Controls	{5}
1.2. Transport Level	{7}
2.1. Datagram	{7}
2.2. Virtual Circuit	{7}
1.3. Higher Level	{9}
3.1. Virtual Terminal (VTP)	{5}
3.2. File Transfer	{8}

TAXONOMY	MILITARY UTILITY
5.4.4. (con't)	
3.3. Command Language	{7}
3.4. Application Specific	{7}
1.4. Controls	{9}
4.1. Establish/Break Connections	{9}
4.2. Addressing/Routing	{9}
4.3. Precedence	{9}
4.4. Acknowledgement	{9}
4.5. Flow Control/Buffer Management	{9}
4.2. Interoperability	{9}
2.1. Message Reformatting	{9}
2.2. Gateways	{9}
2.3. Mail/Message Systems	{5}
4.3. Error Management	{9}
3.1. Detection	{7}
1.1. Cyclical Redundancy Codes	{5}
1.2. Message/Block Serialization	{5}
1.3. Parity Checks	{5}
3.2. Correction	{8}
2.1. (Hamming) Error Correcting Codes	{9}
2.2. Retransmission	{5}
4.4. Security Countermeasures	{9}
4.1. Encryption	{9}
4.2. Antispoofing	{9}
4.3. Playback Detection	{9}
4.5. Network Operation	{9}
5.1. Network Administration	{9}
5.2. Data Collection	{9}

TAXONOMY	MILITARY UTILITY
5.4.4. (con't)	
2.1. Costing	{9}
2.2. Traffic Loads	{9}
2.3. Error Statistics	{9}
5.3. Network Health	{9}
3.1. Diagnostics	{9}
3.2. Restart/Reload	{9}
5.4. Network Processors	{9}
4.1. NFE	{9}
4.2. Concentrators	{7}
4.3. IMPS/TIPS	{9}
4.4. Gateways	{9}
4.5. Security Boxes	{9}

Discussion

Although the military utility of this category is not yet rated in detail, software discussions in the report of TWG-3 (telecommunications) are relevant.

5.4.5. Data Processing Software	{9}
5.1. Operating Systems	{9}
1.1. Batch Commercial	{3}
1.2. Batch/Time-Sharing Commercial	{5}
1.3. Ultrareliable Security Versions (Military/Civilian)	{9}
1.4. Real-Time Military Versions	{9}
1.5. Microprocessor Executives	{?}

TAXONOMY	MILITARY UTILITY
5.4.5. (con't)	
1.6. Operating System for multi-microcomputer and microprocessor configurations	{?}
5.2. Utilities	{?}
5.3. Software to Support DBMS	{6}
3.1. Linear File Systems	{3}
3.2. Hierarchical	{3}
3.3. Network	{7}
3.4. Inverted	{5}
3.5. Relational	{8}
3.6. Query Languages	{?}
3.7. Data Description Languages	{?}
5.4. System-specific *Software* Support Functions	{?}

<u>Discussion</u>

The very high rating {9} has been given for the military real-time operating systems because this software is so specifically intended for military functions. This category is perhaps controlled by the munitions act but is mentioned here for completeness.

The rating {9} for secure operating systems reflects the direct relevance of these systems to tactical C-cubed military functions with a need for multi-level security, for strategic force management applications, and for communications executives (e.g., Autodin II). The high reliability associated with secure operating systems is also of significant military importance. Long mission satellites require high reliability software in addition to high reliability hardware. The desirability of reliability and security in the computers used for the control of ballistic missiles is obvious.

The rating {8} for DBMS software reflects the direct relevance of these systems to C-Cubed military functions and the investment in building such complex software.

5.4.6. Data Storage Software	{9}
6.1. Memory Management	{?}
1.1. Virtual Memory	{?}
1.2. Segmentation	{?}
1.3. Paging	{?}
1.4. Allocation/Reclamation (Garbage Collection)	{?}
1.5. Cache Management	{?}
1.6. Capabilities Management	{?}
1.7. Protection Traps	{?}
6.2. Secondary Memory Management	{?}
2.1. Tape Controls	{?}
2.2. Spooling	{?}
2.3. Disc Control	{?}
2.4. Device Allocation	{?}

<u>Discussion</u>

Time did not permit the military utility breakout for Data Storage Software.

5.4.7. Data Access Software/Data Management System	{9}
7.1. File/Data Management System (DMS)	{?}
1.1. Directory Manager	{?}
1.2. Speech Retrieval Computations	{?}

TAXONOMY	MILITARY UTILITY
1.3. Load Update System	{?}
1.4. Data Semantics Analyzers	{?}
1.5. Data Language Processors	{?}

Discussion

Time did not permit the military utility breakout for Data Access Software.

5.5. Software Development Tools

Software development tools are the next major element of our taxonomy. Other tools, such as those for computer aided design (CAD) and manufacture (CAM) of non-software products, are better addressed by the other TWGs. Integrated systems of tools have a higher military utility than stand alone tools. Systems of tools may be classified in five generic categories:

1. Library Maintenance

These tools include: those used to create directories, files, and versions of files, data base management tools that are used to store and retrieve text from the software library, access control tools employed by the library DBMS, and library tools that interface the library with the native operating system of the hardware for disc and tape access or backup functions.

2. Composition

Tools used to enter, edit, and display/print software text.

3. Translation

These include all integrated tools that translate source text to other source text, or to executable object code.

4. Test and Validation

Tools needed to confirm that the software system is working correctly.

5. Project Management

These are integrated tools used to administer large software projects.

5.5. Software Development Tools {9}

5.1. Software Library Data Base - Used to provide constantly up-to-date representations of the computer programs and test data in both computer and human readable forms. The current status and past history of all code generated are also maintained. Specific library programs are available to serve as aids to implementation. Included are a directory, files, processes, data items and types, and access profiles. {9}

5.2. Software Support Library Management (Data Base Management System) {9}

2.1. Production Libraries {9}

2.2. Retrieval {?}

2.3. Update {?}

2.4. Loader {?}

2.5. Language Processor {?}

2.6. Access Control {?}

2.7. Report Generator {?}

5.3. Text Processing {3}

3.1. Editor {?}

1.1. Text Editor - A computer program used to prepare documentation and perform word-file edits (erase, insert, change, and move words or groups of words). {1}

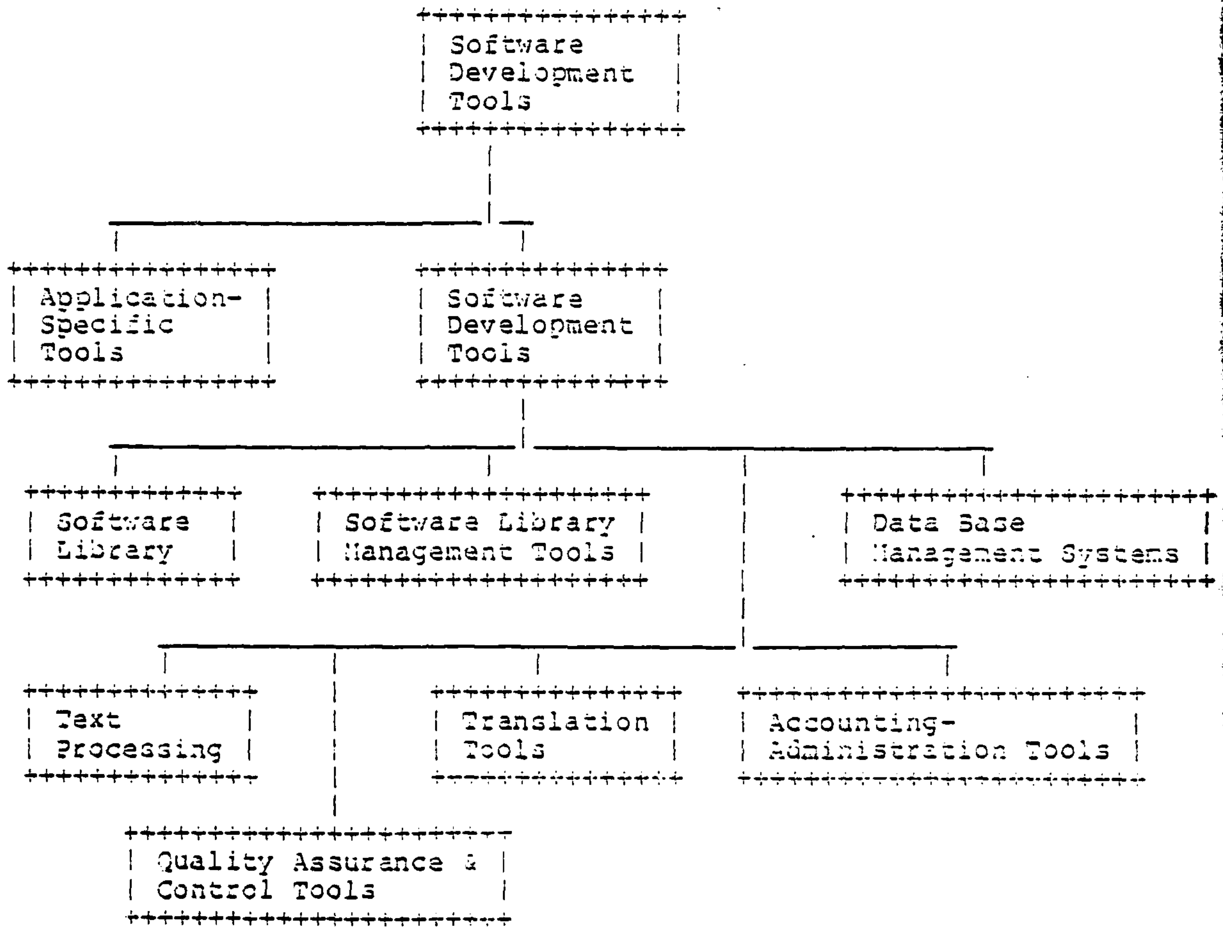

Software Development Tools
Application-Specific Tools
Software Development Tools
Software Library
Software Library Management Tools
Data Base Management Systems
Text Processing
Translation Tools
Accounting-Administration Tools
Quality Assurance & Control Tools

Figure 5.3. Software Development Tools Taxonomy

5.3. (con't)

3.2. Comparator - A computer program used to compare two versions of the same computer program under test to establish identical configurations or to specifically identify changes in the source coding between the two versions. {2}

3.3. Formatter {?}

3.1. Flowcharter - A computer program used to analyze a coded computer program and then to show in detail the logical structure of the analyzed program. The flow is determined from the actual operations as specified by the executable statements, not from comments. The flowcharts which are machine-generated can sometimes be compared to specification-generated charts to show differences. {1}

3.4. Electronic Mail {?}

5.4. Translation {8}

4.1. Higher Order Languages (HOL) {?}

1.1. Compiler - A computer program that either transforms a HOL source program into an assembly language form for subsequent assembly to machine language translation by the assembler, or that transforms directly the HOL program into an equivalent machine language program or potentially into microcode for a micro-programmable computer. {1-8}

4.2. Problem-Oriented Languages (POL) {?}

2.1. Compiler Building/Implementation System - Computer programs that facilitate the development of compilers by use of an HOL and specialized data constructs (e.g., J73-JOC IT). {5}

2.2. Extensible Language Processor - A computer program that allows users to define new language features for extending a base language. {7}

2.3. Hardware Description Languages {8}

5.5. (con't)

4.3. Assemblers (Machine Language) {?}

3.1. Cross Assembler - A computer program that accepts symbolic instruction mnemonics for a selected target computer and generates target computer machine code while hosted on another computer. A cross assembler thus allows code written for one computer to be assembled on another. {1}

4.4. Preprocessors {?}

4.1. Macros {?}

4.2. HOL Syntax Checker {?}

4.3. HOL-HOL Syntax Checker {?}

4.5. Postprocessors {?}

5.1. Compool/Dictionary Builder {?}

5.2. Binder/Linker/Loader/Allocator {?}

2.1. Linkage Editor - A computer program that combines separately produced object or load modules; resolves symbolic cross-references among them; replaces, deletes and adds control sections; generates overlay structures on request; and produces executable code that is ready to be loaded into storage. Also a linking-loader performs the operations dynamically during execution if required. {1}

5.3. Decompiler - A computer program that accepts as data a program written in machine-level language and produces output in higher level problem-oriented target language or the algorithms, etc. of the machine level code. {1}

4.6. Program Libraries {?}

4.7. HOL Translators - Computer programs that are written to convert source code associated with one computer system into equivalent source code on another computer system. {3}

7.1. Compilers {?}

TAXONOMY	MILITARY UTILITY
5.5. (con't)	
7.2. Interpreters - Computer programs that translate and execute each source language statement sequentially.	{6}
7.3. Optimizers	{?}
5.5. Quality Assurance and Control (Test and Validation)	{9}
5.1. Requirements Definitions	{9}
5.2. Instrumentation	{?}
2.1. Automatic Test Generator - A computer program that accepts inputs specifying a test scenario in some special language, generates the exact computer inputs, and determines the expected results.	{6}
2.2. Program Flow Analyzer - A computer program that provides statistics on source code statement usage and timing data on program elements during test case executions.	{5}
2.3. Software Monitor - Computer programs that provide detailed statistics about system performance. They examine such things as core usage, queue lengths, and individual program operation to help measure performance. Data are usually collected mathematically during execution and analyzed later.	{2}
2.4. Timing Analyzer - A computer program that monitors and prints execution times of all program elements (functions, routines, and subroutines.	{8}
5.3. Measurement	{?}
3.1. Analyzer - A computer program used to provide information about some feature of the source program. Usually includes usage of command features and data base items.	{3}

5.5. (con't)

3.2. Data Base Analyzer - A computer program that reports information on every usage of data, identifies each program using any data element, and indicates whether the program inputs, uses, modifies, or outputs the data element. {2}

5.4. Test Cases {?}

4.1. Compiler Validation System - A computer program used to ensure that compilers meet their language specification. {4}

4.2. Test Case/Test Data Generator - A generator that produces test data or test cases to exercise the target system. A generator differs from a simulator, because it creates test data using numerical generators, etc. Once the data are produced by the generator, a simulator might be required to route the data to the system. {7}

5.5. Synthetic Data/Load {?}

5.6. Emulator/Simulator {?}

6.1. Emulation - The use of programming techniques to permit a computer system to execute object level programs written for a given computer. Sometimes confused with an interpretive instruction simulator with execution, in a standard machine, of the object code. {3}

6.2. Instruction Simulator - A computer program used to dynamically simulate the execution characteristics of a target computer using a sequence of instructions of a host computer. {5}

5.7. Debugger - Compile and execution-time checkout and debug capabilities that identify and help isolate program errors. {5}

7.1. Breakpoints

7.2. Symbol Dictionary {?}

7.3. Code/Data Reference Traps {?}

7.4. Code Flow Traces {?}

5.5. (con't)	
4.1. Instruction Trace - A computer program used to record every instance of the occurrence of a certain class of operations or a given set of conditions or to trigger event-driven data collection. In some cases, this creates a complete time record of every event occuring during program execution.	{2}
7.5. Conditional Data Dumps (Snapshots)	{?}
5.1. Snap Generator - provides program or data locations that are relative to program labels. Used typically to present a picture (data contents) of a selected portion of memory.	{1}
7.6. Try-Exit Conditional Code Execution	{?}
7.7. Conditional Backup and Retry	{?}
5.8. Program Verification Systems - Programs that verify the correctness of software either by formal proof or by instrumenting the source code. They provide data that shows how thoroughly the source code has been used. Examples are RXVP, JAVS, PET.	{9}
8.1. Initial Conditions	{?}
1.1. Program Sequencer - A computer program which coerces the execution of all possible instructions and branches within a program to determine program flow, to execute seldom-used branches, and to assist in the verification of proper program operations.	{6}
8.2. Correctness Criteria (Invariants)	{?}
8.3. Specification and Design HOL and Processor	{?}
3.1. Consistency Checker - A computer program which determines (1) if requirements and/or design specified for computer programs are consistent with each other and their data base and (2) if they are complete.	{6}

TAXONOMY	MILITARY UTILITY
5.5. (con't)	
3.2. Design Language Processor - A computer program used to provide an understandable representation of the software design as it evolves. These programs allow designs to be constructed and are expanded in a hierarchical fashion.	{3}
3.3. Interface Checker/Analyzer - A computer program that is used to automatically check the range and limits of variables as well as the scaling of source programs to assure formal compliance with interface and control documents.	{5}
3.4. Requirements Language Processor - A computer program used to provide a succinct and unambiguous specification of the system based on computer requirements. It allows requirements to be communicated and translated in a hierarchical or other organized manner.	{9}
3.5. Requirements Tracer - A computer program used to provide traceability from requirements through design and implementation of the software products.	{9}
8.4. Verification Condition Generator (VCG) for HOL	{?}
8.5. Theorem Proving System	{?}
8.6. Configuration Management (Spec, Code, Proof Text)	{?}
5.9. HOL Syntax-Semantics Checkers	{?}
9.1. HOL Editor - A computer program used to analyze source programs for coding errors and to extract information that can be used for checking relationships between sections of code.	{4}
9.2. Code Auditor - The editor will scan source code and detect violations to specific programming practices and standards, construct an extensive cross-reference list of all labels, variables, and constants, and check for prescribed program formats.	{4}

TAXONOMY	MILITARY UTILITY
5.5. (con't)	
5.6. Accounting/Administration	{9}
6.1. Software Configuration Control System	{9}
1.1. Standards Enforcer - A computer program used to determine automatically whether prescribed programming practices and standards have been followed. The program can check for violations of standards set for such conventions as program size, commentary, structure, etc.	{5}
1.2. Structure Analyzer - A computer program used to examine source code and determine that structuring rules, set for either control or data structures or both, have been obeyed.	{2}
1.3. Computer Program Management Aids	{9}
3.1. Cost Estimating Programs for Estimating Cost of Programs	{3}
3.2. Time Estimating Programs (Development/Production time)	{5}
3.3. Configuration Management Systems	{9}
6.2. Engineering Change Control (ECC)	{?}
6.3. Access Control	{?}
3.1. User Access Profile	{?}
3.2. Security	{?}
3.3. Privacy	{?}
3.4. Accountability	{?}
6.4. Safety	{?}
4.1. Backup	{?}
4.2. Recovery	{?}
4.3. Data/Command Legality and Reasonableness Checks	{?}
6.5. Cost Controls	{?}
6.6. Methods and Procedures and Supporting Documentation	{?}

5.6. The Feasibility of Controls Based on a Taxonomic Approach

After considerable deliberation, the software subgroup has come to the conclusion that a taxonomic approach to export control for systems software is seriously deficient in at least two ways.

First, such an approach does not directly address software life-cycle development and maintenance technology, which we believe to be the most critical issue (see Sections 2 and 3). A taxonomy and military utility evaluation, such as that given for systems software in Sections 5.3. and 5.4., focuses attention on products rather than on technology and technology transfer.

Second, even as a means for the analysis of products and some of the technology embedded in these products, such an approach is too coarse and unmanageable. The basic problems are that software use is extraordinarily pervasive and that there is an enormous range of software products. In Figure 5.1. and in Section 5.4 we have two taxonomies for systems software. A "first level" breakdown is given in Figure 5.1. At this level of detail, a military utility evaluation produces very high ratings for virtually every category. The reason is that, within each systems software category at this level, it is possible to identify important military use. In Section 5.4., we exhibit a "second level" breakdown which is an order of magnitude longer and more detailed than that of Figure 5.1. At this level of disaggregation, we are able to identify quite a few categories with military utility ratings of less than 7, although many continue to have ratings from 7 to 9.

The "third level" breakdown, which we have not done, would explicitly consider specific named products, portability, etc. It would be an order of magnitude larger and more detailed than our "second level" breakdown, i.e. about 100 pages long. At this level of disaggregation, we would find that most software products need not be controlled. For example, under Section 4.5.2.2., Human Data Input, we find natural language and speech input given high ratings. One can identify or imagine important military software systems in these categories that should be given high ratings. However, one can also imagine academic or commercial systems in these categories that could be safely sold to users in adversary countries with suitable safeguards (e.g., object code and user manuals only, and with a lack of portability - see Section 6.5).

The problem with giving licensing officers, or government personnel who are asked to evaluate license requests, "control categories" like "real-time operating systems" at the first or second level breakdowns is that this is too coarse. They either have to say "no" to all requests with items in such categories, or they have to work out a third level disaggregation. Even with this level disaggregation, this approach encourages an attitude of "we cannot sell them anything they do not already have." Regardless of one's political philosophy, we can all understand why customers in adversary countries would not be interested in spending much hard currency for what they can get from indigenous sources. The software subgroup feels that this is counter to U.S. academic and commercial interests, that it is potentially an enormous burden for the U.S. government, and it is neither the only nor the best way to protect U.S. national security with respect to software technology transfer.

We feel that a taxonomic approach for applications software would also be too coarse and unmanagable. Any attempt to provide a taxonomy of applications software would be at least an order of magnitude larger, at each level of detail, than the corresponding taxonomies for systems software. However, we have isolated four classes of software which we believe should be controlled because of their high military criticality. Software development tools, in particular, may be controlled using a partial taxonomic approach.

6. Software Technology Transfer Mechanisms

6.1. Introduction

This section presents a taxonomy of software transfer mechanisms, an evaluation of their effectiveness, and an examination of the feasibility of controlling some of these mechanisms. A classified appendix looks at some mechanisms in more detail than would be possible in the main body. This appendix has been detached from the main body of this report and is available through appropriate channels.

6.2. A Taxonomy of Transfer Mechanisms

The taxonomy presented here is a more detailed examination of the general breakdown of transfer mechanisms given in Section 4. The transfer mechanisms can be divided into three groups: assistance, product shipment, and usage. Covert mechanisms which parallel many of the transfer mechanisms in the above categories are also examined.

6.2.1. Assistance

The assistance type of transfer mechanism includes a variety of services, training, and education, and often involves the interaction of technical personnel. Assistance is divided into product support assistance and general assistance. Figure 6.1 presents a taxonomy of assistance mechanisms in a tree structure format.

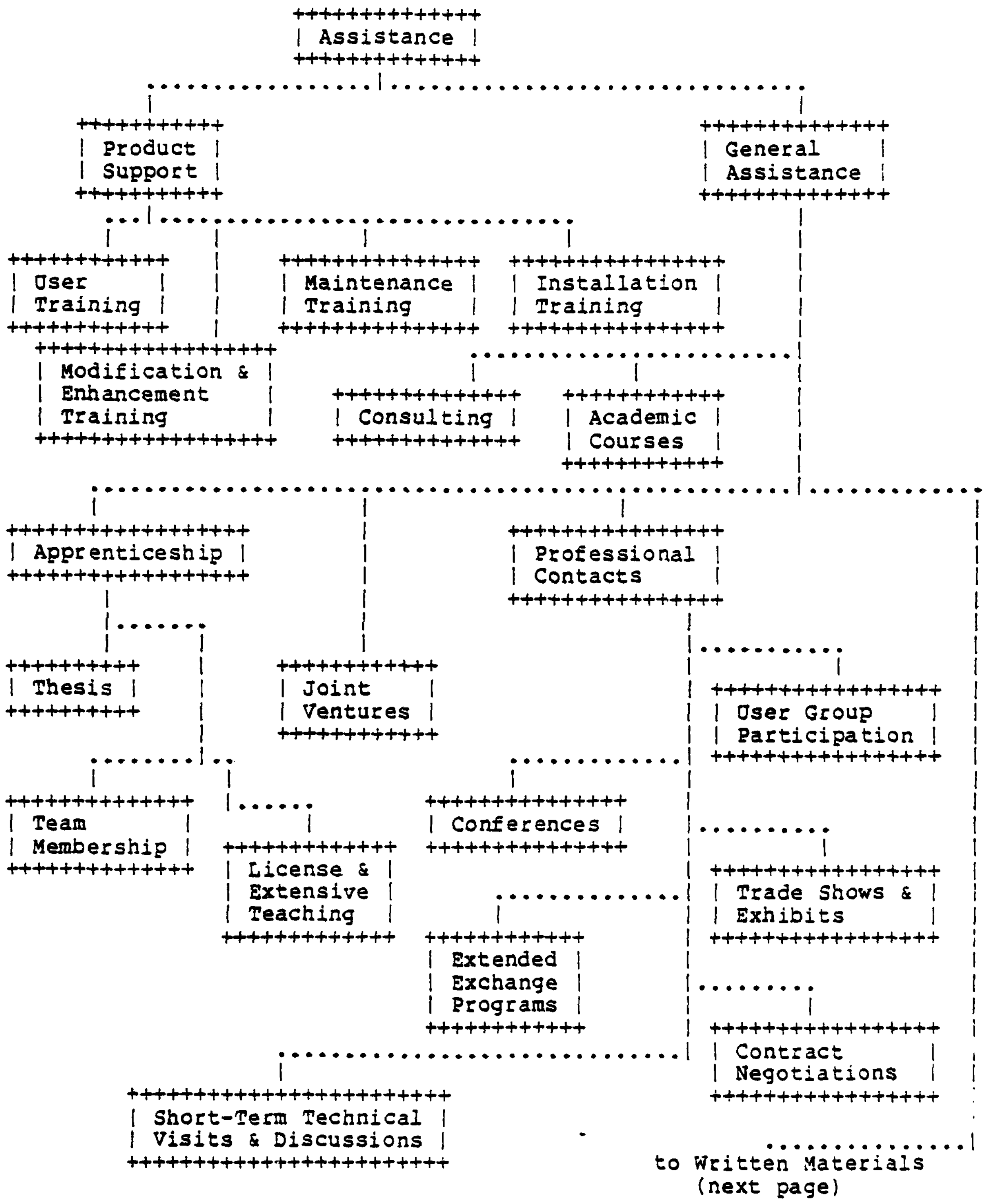


Figure 6.1. Assistance Mechanisms

```
       .................(from preceding page)
       |
       |
 +++++++++++++
 | Written   |
 | Materials |
 +++++++++++++
       |
       |............................................
       |                 |           |            |
       |           +++++++++   ++++++++++++   +++++++++
       |           | Books |   | Journals |   | Other |
       |           +++++++++   ++++++++++++   +++++++++
       |
++++++++++++++
| Courseware |
++++++++++++++
       |
       |............................
       |          |                |
 +++++++++  ++++++++++++++   +++++++++++++++
 | Texts |  | Programmed |   | Computer-   |
 |       |  | Learning   |   | Aided       |
 |       |  | Texts      |   | Instruction |
 +++++++++  ++++++++++++++   +++++++++++++++
```

Figure 6.1. Transfer Mechanisms (con't)

6.2.1.1 Product Support Assistance

This typically takes the form of training or other support services related to a particular product. The types of training that fall into this category are: (1) user training; (2) installation training; (3) maintenance training; and (4) training to make modifications and enhancements of the product. Product support services also involve some assistance with software problem determination, provision of fixes, and guidance in system usage.

6.2.1.2. General Assistance

This mechanism may be broken down into five categories:

(1) Apprenticeship--Any relationship involving strong collaboration between a Western expert and an adversary "student" from an adversary country, or groups of these in some joint work. Examples are thesis research, team membership in a software development project, and a license with extensive teaching.

(2) Joint ventures--These relationships also involve strong collaboration, but differ from apprenticeships in that the participants may be transferring technology in both directions. Joint ventures may have various degrees of coupling.

(3) Consulting--Western experts, both as individuals and groups, provide specific consulting services for adversary countries. They may provide advice, produce a piece of work (i.e., develop or tailor a software package in accordance

with customer specification), or provide support services which do not relate to a specific product which the consultants have provided.

(4) Academic courses--Students from adversary countries enter Western academic computing courses provided by university, government, or industrial organizations, but do not advance far enough to be considered apprentices; or instructors from the West go to adversary countries to teach such courses.

(5) Professional contacts--This category includes a wide variety of transfers such as extended exchange programs (but not at the level of categories (1) and (2)), short-term technical visits and discussions, participation in user groups, trade shows and exhibits, conferences, and contractual/commercial discussions.

(6) Written material--This category includes all open and proprietary passive sources of information such as textbooks, journals, articles, etc., as well as courseware: programmed instruction texts, computer-assisted instruction, and other materials which may be of academic, governmental, or industrial origin.

6.2.2. Product Shipment

The product shipment transfer mechanism is distinguished by the delivery of certain physical items (not including product support assistance and written material as described in category (6) above). Software products and documentation come in many forms. Some examples are proposals in various levels of detail, design specification data, intermediate design documentation, program logic manuals, users manuals, code listings in HOL, assembly language, or object code form, and source and object code representations in such physical forms as tapes, disks, floppies, cards, read-only memory, etc. Figure 6.2 presents a taxonomy of product shipment forms.

6.2.3. Remote Access Usage

This category refers to providing a user from a adversary country with access to the usage of software and data via some kind of telecommunications link (e.g., a computer network or phone line). The hardware and software is owned and controlled by the West.

6.2.4. Covert Mechanisms

The covert acquisition of operational capability or know-how can be divided into the same three categories listed above. Means of getting assistance might include infiltration of American or subsidiary companies by individuals, dummy corporations set up to buy American products with all the associated product support assistance, compromise and blackmail of American technical experts, acquiring documentation and other aids from the development process, etc. Physical products may be purchased through other agents, stolen, or legally purchased in pieces and assembled. A particularly good way to transfer both know-how and operational capability would be to purchase part of an American software corporation through a front organization. Furthermore, many third countries may serve as conduits. For all the mechanisms the software subgroup can postulate, adversaries are likely to come up with others.

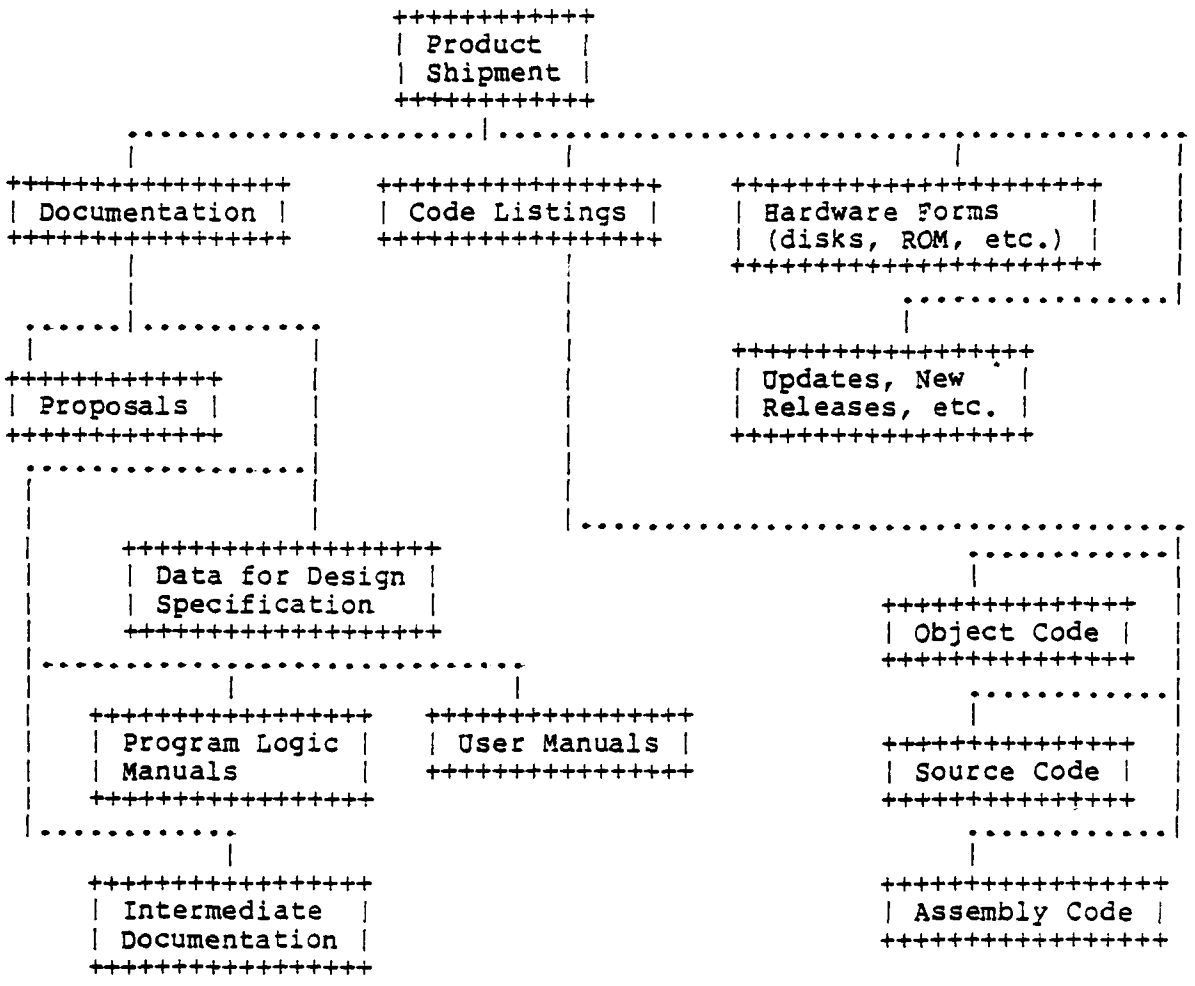


Figure 6.2. Product Shipment Mechanisms

6.3. The CNCTEG Framework for Evaluating the Effectiveness of Transfer Mechanisms

It proved impossible to develop a linear ranking of the effectiveness of various different mechanisms for software technology transfer. There are simply too many cases in which other factors may alter the order of any such ranking. Instead, the software subgroup has found it useful to consider transfer mechanisms in light of four general factors developed by the CNCTEG.

6.3.1. Know-how Transfer

Two factors influence the effectiveness of the transfer of know-how. The first is the nature of the know-how transfer mechanism. At one end of the spectrum are "active" mechanisms in which frequent interchange of information exists, e.g., an individual ("student") working with a vendor's programming team ("teacher") as a regular employee for some period of time. "Active" mechanisms tend to be iterative in nature; "students" seek information, receive answers, and seek further information. At the other end are "passive" mechanisms in which no interchange exists, e.g., an individual ("student") disassembles a program and derives or infers certain facts about the design strategies underlying the item. The terms "active" and "passive" refer to "teacher" activity; adversaries may very actively seek and use information they get from passive sources. "Active" mechanisms are normally of more concern than "passive" mechanisms because they tend to transfer know-how as well as operational capability.

The second factor is the kind of know-how transfer involved.

At the lowest level of effectiveness are those transfers which deal only with operational information, such as how to use a product and make minor modifications. The medium level includes the information of the first level and adds specific knowledge about structure and design, e.g. specific information about interfaces between modules. The highest level includes the first two, but also includes explicit knowledge about how the program was produced, i.e., the life-cycle management technology. As the spectrum is traveled from the lowest level to the highest, the information added is more likely to give the adversary the ability to build, modify, adapt, and maintain programs.

6.3.2. Operational Capability Transfer

Two factors influence the effectiveness of the transfer of operational capability. The most important is the nature of the operational capability. At the highest level of transfer effectiveness are offerings that directly provide operational capability of military concern, e.g., a nuclear weapons design program. At the middle level of transfer effectiveness are offerings that provide an appropriate base that may be modified at a cost significantly less than starting from scratch to provide the specific application operational ability of concern, e.g., a civilian air traffic control system that could be modified to meet military requirements. At the lowest level of transfer effectiveness are offerings that provide operational capability of no concern, e.g., a payroll program.

The second factor involves the form of the operational capability, which may range from "easy to replicate" (e.g., a program recorded on a reel of tape) to "hard to replicate" (e.g., a program embedded in a semiconductor chip). Also, the product may or may not be in a form which can be easily used on an adversary's computer. The "easy to replicate" forms of operational capability are of much more concern because they potentially transfer an unlimited quantity of the operational capability in question, given that suitable systems and supporting services are available.

These four criteria may be applied to each of the transfer mechanisms outlined above. In most cases there is a wide range of possible levels of effectiveness; the software subgroup has isolated those combinations which are of greatest concern.

6.4. Evaluation of Transfer Mechanisms

6.4.1. Apprenticeship

Many of the types of apprenticeships listed above are likely to be highly effective because they are very active, i.e., a "student" works under the close supervision of a knowledgeable team or person [9]. This kind of experience-building, iterative contact is not available via passive sources but is essential to learning how to put together large pieces of software that work. Any consideration of evaluating apprentice relationships should be in terms of how much participation in the social process of life-cycle software development they permit. A thesis project on a highly theoretical subject, for example, which involves only a single teacher and student is not likely to be of concern.

6.4.2. Joint Ventures

Joint ventures with full participation of both sides in the development of a product are likely to transfer significant amounts of know-how in the development stages, and possibly high degrees of operational capability as well. Such a venture will probably result in a product which is well-suited for its application, and participation in the development process makes maintenance and adaptation less difficult. Joint ventures which glue together separate research and development efforts include almost as much potential for technology transfer, since a substantial amount of joint work is required to make such a product work in practice.

6.4.3. Product Support Assistance

The product support mechanism may involve an "active" relationship between "student" and "teacher," and therefore has the potential for substantial transfers of know-how. However, this relationship differs from that of an apprenticeship not only because it involves a single product, but also because the "teacher" may have a strong incentive to prevent large transfers of know-how which would make the "student" independent (or even competitive) in the future.

Product support assistance which is of greatest concern is that dealing with installation, maintenance, and enhancement, since these kinds of training raise the adversary's ability to make the product more portable and to adapt it to other purposes. Of most concern is training which imparts general O&M skills. User training which transfers little more than operational know-how (for example, a limited amount of classroom instruction) would not be of great concern.

6.4.4. Consulting

A wide range of activities fall under this heading. At one extreme are "student"-"teacher" relationships which function as apprentice relationships as outlined above. At the other extreme are consulting activities which involve solving a specific problem for the adversary with minimal participation by him, and which reveal little about the solution or the social process of producing it. Therefore, the consulting activities which are of concern are those which are active, which allow adversary participation in the solution process, and which transfer O&M capability.

6.4.5. Product Shipment

The effectiveness of the product shipment mechanism depends on the physical forms of the transfer. Various documentation forms (e.g., design specification data, intermediate documentation, user manuals, and program logic manuals) and source code listings alone or together can transfer a substantial amount of know-how and operational capability. The effectiveness of product shipments can be greatly enhanced by the presence of arrays of these materials or other sources of information. Linking together a detailed manual on software engineering, design specifications, intermediate documentation, and the final source code product, for example, may result in the transfer of substantial amount of know-how. Therefore, particular care should be taken to consider what arrays of products are being shipped.

One of the hypotheses of the software subgroup has been that, all other things being equal, software which is portable and can easily be reconstructed should be of more concern than software which does not have these characteristics. Consequently, there should be ways of protecting software so that some operational capability can be transferred without transferring a substantial amount of know-how or an open-ended operational capability. What are needed are effective means by which the software can protect its own "secrets" and prevent misuse. Some suggestions for such means will be considered in Section 6.5. For now it is enough to examine methods presently in use.

The traditional means used by Western corporations for controlling the unauthorized replication, distribution, and modification of software, and for preventing transfers of proprietary information contained in a product, have been two. First, they have relied on contracts and traditional legal sanctions to enforce regulations about unauthorized duplication of programs which need no modification and can be used at any installation. Second, for programs which need modification, they have relied on selling only object code versions of programs. This is a form of weak encryption which has been relatively effective because American corporations have seen little value in spending the resources needed to decompile object code. Can these traditional means be effective control mechanisms against the adversary countries?

The control mechanisms used in the U.S. probably are not as effective with regard to adversary countries. In the first place, sanctions against unauthorized replication, duplication, and modification cannot be enforced. Contracts and sanctions have proven to be almost worthless internationally. Secondly, it is very difficult to detect when a specific violation has occurred, especially if a product has been transferred to a military installation. Finally, the form of the product may or may not be an effective barrier against transfers of know-how.

The CNCTEG reached the tentative conclusion that the form of code transferred, i.e. source versus object, did not make a great deal of difference with respect to the transfer of operational capability (neither form alone can transfer a significant amount of know-how). The TWG-7 software subgroup has come to modify that conclusion for a variety of reasons. Although it remains true that source code versions of programs can be obtained in some instances with relative ease and that execution environments can be emulated, the availability of only object code can effectively hinder technology transfer in certain respects.

There are limits to how much information can be obtained from an object code version of a program. In the first place, decompiling is technically imperfect and can only produce code without comments. Much decompiling is really de-assembling; that is, creating semi-symbolic assembly language source listings of the CPU operation code symbolics only. There are few, if any, symbolic data or address references, and almost never HOL symbolic source statements output from (known) decompilers today.

In the case of modern optimizing compilers, there is some doubt as to whether decompiled code will even vaguely resemble the source code form or even be comprehensible. Furthermore, decompiling can offer little insight into the relationships between modules (who "talks to" whom) which is such an important part of large integrated systems. A fully-documented source code program can yield insights into the design process, especially when used in conjuction with "how-to" materials, while an object code program which has been decompiled cannot. Some two-thirds of software development costs currently go towards maintenance and enhancement, yet in many cases, having object code alone would not be sufficient to carry out these activities, or would at least make them very difficult.

The Soviets have probably had much more experience with decompiling than has the U.S. They do not face the same legal and economic restraints present in the U.S. They have a large pool of programmers who have actually been trained in machine language skills, and have shown a desire to borrow from the West even when they may possess the skills necessary to do it themselves. As has been noted in Section 3, the Soviets are on the threshold of a new period of software development. The form and kinds of products acquired by them from the West may have a substantial impact on the course this development effort follows. There is no guarantee that the Soviets are unwilling to devote the resources needed to successfully decompile large programs. Apparently, the Soviets are willing to go to some length to obtain source code[10].

The provision of updates, new releases, etc., should be considered at the same time as consideration of the sale of the product itself is made. The physical forms of updates, new releases, etc., should not differ from the forms of the original product.

6.4.6. Academic Courses

The desirability of controlling this mechanism depends upon the degree of "activeness" of the relationship between "student" and "teacher" and the nature of the technology being transferred. Some well-motivated foreign "students" cultivate a very active relationship with the "teacher." The academic courses which are of most concern are those that present information which is not available in the passive literature and include projects which impart a significant amount of know-how. For example, seminars offered within a company which involve proprietary information should be of great concern.

Another important factor is what information and resources are accessible to the "student." While it might seem unnecessary to control an introductory computer science course, if such a course gave the "student" unlimited access to computer center resources for an extended period of time, substantial know-how and operational capability transfers could take place. Even the resources of a good library and access to a Xerox machine could result in substantial transfers given enough time. Furthermore, if the course itself deals with software modification, adaptation, and enhancement, it could impart to the student significant skills in these areas.

6.4.7. Professional Contacts

Professional contacts involve a wide range of interactions between adversary seekers of information and Western sources. Hence, the effectiveness of these mechanisms enjoys perhaps the widest variations of any of those outlined here. Brief encounters via a conference, user group participation, correspondence, even contract negotiations may result in an active relationship which could result in a transfer of know-how. Furthermore, sizeable amounts of know-how may be transferred by a series of brief encounters which, by themselves, are unimportant. The amount of information transferred, the replicability of any products acquired, and the suitability of those products also can vary widely according to the kind of relationships which develop. Even a short description of a new research direction in the West may save the adversaries substantial resources by indicating the directions they should pursue.

Given the difficulty of assessing how effective this mechanism may be, the software subgroup has reached the conclusion that the criterion of time should be adopted as one means of realistically evaluating the kinds of transfers which can take place via this mechanism. Exchange programs and other professional contacts of an extended nature (a month or longer) are of most concern. The classified Appendix to this section discusses some forms of this mechanism in more detail.

6.4.8. Written Materials

Some forms of written sources may result in significant technology transfers. For example, modern computer-assisted instruction routines, programmed learning texts, or other "courseware" may be effective. Other publications which discuss state of the art developments can also be extremely useful to adversaries.

The software subgroup has found that there is a substantial amount of information which is of concern that is available through governmental and academic channels. However, the wide variety of materials available, the difficulty of devising control mechanisms, and the absence of supervision and iterative feedback in its use imply that only classified or proprietary information can or should be controlled.

6.4.9. Remote Access Usage

While remote access usage generally poses little threat of technology transfer, certain types of software may permit substantial technology transfers to occur. Modern interactive software products are designed to help the user every step of the way. A determined adversary "probing" such a system may be able to learn a substantial amount about its inputs, outputs, limitations, etc. Furthermore, software systems are beginning to incorporate expert insights in various fields. Software which is intended to help a user create and design such expert systems would carry a high potential for technology transfer if used by an adversary. Since this software has the ability to behave like a knowledgeable human being, active transfers can take place even though a human is not present. Another concern presented by remote access usage is that an adversary user might gain unauthorized access to software or data.

6.4.10. Covert Mechanisms

There are several factors which influence the effectiveness of covert mechanisms. First, the adversary must be able to define specifically what he wants to acquire in the American "marketplace." While this is not always an easy matter, past experience has shown that seemingly innocent technical visits by trained personnel have served as a means for "shopping" for later acquisitions. If the adversary is ready to risk a covert acquisition, he is likely to obtain programs which have a high operational value and can be easily replicated.

Once the acquisition requirement has been defined, the number of intermediaries between those requesting the product and those acquiring it becomes important. Clearly, if the acquisition of a product is entrusted to someone who really has a limited conception of what to take, the likelihood that all of the required components will be acquired is diminished. Furthermore, it would be quite useful to be able to make an on the spot assessment to see what complementary products, such as documentation and maintenance aids, might be available with the product. Finally, the product which is acquired must have a suitable hardware host in which it can function. The advent of a suitable hardware base in the Ryad and SM models may be spurring an increased desire for covert acquisitions.

One argument which the software subgroup rejects is the idea that exports should be permitted on the basis that the "adversary can get it anyway" via covert means. As it should be clear from

AD-A106 869   INSTITUTE FOR DEFENSE ANALYSES ARLINGTON VA SCIENCE A--ETC   F/G 9/2
SOFTWARE TECHNOLOGY TRANSFER AND EXPORT CONTROL.(U)
JAN 81   S E GOODMAN, N S GLICK, W K MCHENRY   MDA903-79-C-0018
UNCLASSIFIED   IDA-N-878   IDA/HQ-81-23408   NL

2 OF 2
AD
A106869

END
DATE
FILMED
12-81
DTIC

the above, the effectiveness of such a transfer will probably be far less than that of an outright sale, especially with a number of intermediaries in the acquisition chain.

6.5. A Partition of Transfer Mechanisms for the Purposes of Export Control

To attempt to control all of the transfer mechanisms outlined above would be highly undesirable and impossible in practice. The software subgroup has partitioned the mechanisms into four general categories. The first category contains some of the most effective mechanisms for which adequate controls may be possible. The remaining categories contain mechanisms that are less effective, or harder to control, or for which controls are less desirable (e.g., because controls may hurt us more than they hurt an adversary). Such a partition does not attempt to address the feasibility or desirability of controlling every technology transfer; rather, it is intended as a broad, tentative guideline only. Furthermore, with the partial exception of the section on technical measures below, the subgroup has not considered how these controls should be implemented. Such a task is beyond the resources and the purposes of the subgroup.

Some undesirable transfers take place merely because of ignorance on the part of United States citizens about the technological levels and interests of an adversary. It is our hope that our study will alert the United States data processing community to potentially damaging transfers. The software subgroup holds the view that voluntary restraints by an educated data processing community may be at least as effective as government controls.

6.5.1. Categories

The following transfer mechanisms apply to all transfers, regardless of source (industrial, academic, or governmental).

<u>Category I</u>

Effective mechanisms for which adequate controls may be possible.

Apprenticeships:	Team Membership
	License with Extensive Teaching
Joint Ventures	
Product Shipment:	Development Data Base
Product Support Assistance:	Modification and Enhancement Training
	Maintenance Training
Consulting	

<u>Category II</u>

We feel that these mechanisms are less effective and more difficult to control than those in Category I, but that control may be possible.

Professional Contacts:	Extended Visits (without apprenticeships)
	User Group Participation
Product Shipment:	Source Code and Detailed Documentation
Product Support Assistance:	Installation Training
Academic and Other Courses:	State-of-the-art training (including courses), not widely available

Contract Negotiations:	Detailed negotiations and proposals
Usage	

Category III

While control of the following mechanisms may be possible, it is doubtful if such controls would be desirable or worth the effort of imposing them.

Professional Contacts:	Control of visiting adversary nationals' movements in those parts of host organization's facilities that are accessible to all members of that organization (e.g., restricted access to computer centers at commercial or non-profit organizations).
	Movement of U.S. citizens abroad
	"Licensing" people who deal with adversary countries in technical matters

Category IV

We feel that control of the following mechanisms would be highly undesirable either because they only involve weak technology transfers or because controlling them would impose severe constraints on the software industry in the United States.

Written Material:	Widely Available Literature (not classified or proprietary)
	Correspondence
Product Support Assistance:	User Training for Purchased Products
Product Shipment:	Object Code and Users Manuals
Professional Contacts:	Short Term (less than two weeks) Contacts and Visits

	Short Term Visits to Trade Shows, Exhibits, and Conferences
	Work that is Primarily of a Theoretical Nature
Apprenticeships:	One-on-one Thesis Supervision (but restrict exposure to facilities or group projects)
Academic Courses:	Courses that are Widely Available

6.5.2. Technical Measures

A promising approach to the problem of source versus object code sales and problems of portability, replicability, etc. is the use of technical measures to tie software to the particular machine on which it is being run and to ensure that the software itself is not tampered with.

What forms should these technical measures take? One possibility would be to have the software read special hardware "signatures" which would be unique to each computer. If the software could not make a correct reading, it would cease to function or produce intentionally wrong results. Buried in thousands of lines of object code, such a command would be exceedingly difficult to find, perhaps entailing about as much effort as reverse engineering the entire system. For stand-alone software sales, the software could be tuned for the computer it is to run on by measuring certain timing or other unique characteristics of the machine. Software can also be transferred in hardware forms such as ROM.

These ideas should be taken as nothing more than promising suggestions. It is our belief that government-funded research in this area could produce simple, creative measures which would not only be an an effective aid in limiting technology transfers, but also a means of preventing bootlegging and unauthorized transfers of software in the United States. As software investment continues to burgeon, U.S. manufacturers will have increasing incentives to use such measures. Furthermore, the dramatic reductions in the cost of hardware predicted for the future will make such technical measures economically feasible.

7. Some Thought Experiments

7.1. Introduction

In this section, we try to pull together the "Who", the "What", and the "How" of software technology transfer. We try to view the acquisition of software and the associated process from the perspective of a determined antagonist. This antagonist has a number of reasons for wanting to acquire software and information about it. One is to determine Western capabilities in order to establish where we stand with respect to the state of the art in a given area. While this information is useful for intelligence and counter-command purposes, it is not an area of concern of this section. Other major reasons for acquiring software, as opposed to developing it from scratch, are to reduce the expenditure of resources - some of which, like system programmers, may be in critical supply - and to reduce the amount of time that it normally takes to develop and field a tested system.

Basically there are two approach_s that such an antagonist might pursue to acquire the software and related documentation. The first approach is to determine the life-cycle of the software of interest, to identify the phases (See Section 2) and to attempt to acquire as much as possible of the project library data base produced during each life-cycle phase. The second approach uses a functional analysis to determine the requirements which must be satisfied by the software in question and then attempt to fulfill those requirements by acquiring commercially available software.

7.2 Software Life-Cycle Phases

7.2.1. Phases and Products

The normal DoD software life -cycle phases are five: conceptual validation, requirements validation, full-scale development, production, and deployment. Within this cycle, three key decision points are reached: (1) Program Decision - following the conceptual phase; (2) Ratification Decision - following the requirements validation phase; and (3) Production Decision - following the full-scale development phase. These decision points are supported by the Defense Systems Acquisition Review Council (DSARC) and are designated in Figure 7.1 as DSARC 1, 2, and 3.

These DoD phases include all of the generic life-cycle phases outlined in Section 2.1. The conceptual phase includes the process of concept definition; from this phase an initial specification is produced. A complete requirements document results from the requirements phase. The four software life-cycle functions of design, coding and checkout, testing, and integration occur during the full scale development and initial production phases.

The major milestones that occur during the system life-cycle phases and their associated documentation are indicated in Fig. 7.1. In the requirements definition phase, the final system specification is issued. Also the draft part I (development) and the draft interface control drawings (ICDs) are prepared and reviewed at the system design review. Quite often, a computer program configuration item (CPCI) requirements review will be held at the end of the requirements definition phase.

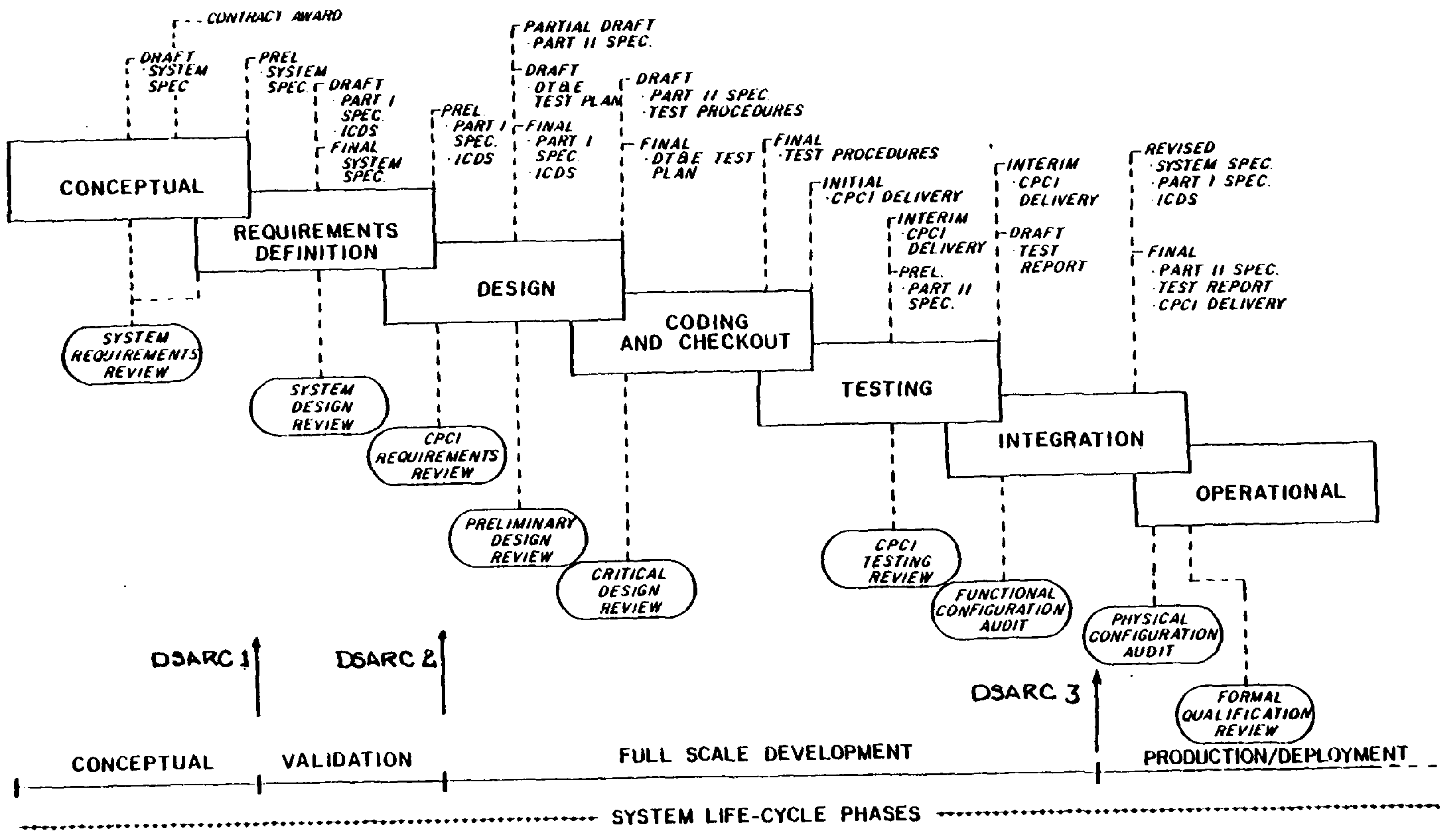


Figure 7.1 Software Life-Cycle Phases and Major Milestones

The final part I specifications and ICDs are issued in the design phase. Preliminary test plans and partial draft II (product) specifications are prepared for review at the preliminary design review (PDR). The design phase ends with the issuance of the final test plan, the draft test procedures and the draft part II specifications, which are reviewed at the critical design review (CDR).

In the coding and checkout phase, the test procedures are finalized and the initial CPCI delivery is made. In the testing phase the interim CPCI delivery is made and the preliminary part II specification is issued. During the integration phase the final copies of part II specification, the test reports, and the CPCIs are delivered. Revised copies of the system and part I specifications and the ICDs are issued incorporating the approved changes so that current documentation is available for transition to the operational phase. The physical configuration audit (PCA) is held following all the revisions and updates.

It should be recognized that Figure 7.1. illustrates the idealized flow through the cycle. In reality there are numerous feedback iterations which occur due to requirements or design changes, hardware/software tradeoffs, problems encountered during testing, etc.

7.2.2. Technology Transfer Utilizing Life-cycle Products

An adversary wanting to improve his own operational capability and at the same time desiring to reduce expenditures of critical resources (including calendar time) might choose to do so by acquiring a copy of a Western system. One way to do this would be to attempt to acquire all of the documentation products indicated on Figure 7.1 (the project library data base). Even on a classified project, a surprising percentage of the documents may not be classified, so that accessibility by an adversary may not be effectively restricted.

If all the products are not accessible, it is a useful exercise to assess the impact on resource expenditures if only certain of the products are available. In order to do this, let us hypothesize a typical software system project, project A. We assume that the size of the delivered software system is 100,000 lines of source code, that the human resources expended were 500 person-months of effort and that the length of the development cycle (from requirements analysis through system acceptance) is 24 months. Figure 7.2. shows the resources expended during each of the four main phases of the system life-cycle and the elapsed time required in each of these phases.

Phase	Person-Months [% of total]	Elapsed Time
Requirements & Specifications	50 [10%]	3 months
Design	125 [25%]	5 months
Coding & Checkout	125 [25%]	9 months
System Integration and Testing	200 [40%]	7 months

Figure 7.2. Human Resources and Time Requirements Distribution for a Typical Software Project

From this example, we see that if an adversary could acquire only the requirements and specification documentation, he would be able to reduce his resource expenditures by no more than 10% and reduce his development cycle three months. For purposes of simplicity we assume that there is no loss of information in transferring, reading, and understanding the documentation, nor are there any resources expended by the adversary on learning and "coming up to speed" on the project. As a matter of fact, as more software systems are engineered using modern technologies, such as formal specifications, hierarchies of abstract machines, etc. the more this simplifying assumption becomes true, and the more resources can be saved by an adversary who captures the early requirements and specification documentation.

If the adversary could acquire the requirements and design documentation, he would reduce his expenditures of resources on the project by as much as 35% and reduce his development time by perhaps eight months. If he could acquire the checked out code and its documentation, he might reduce his resource expenditures on the project by 60% and his development time by 17 months.

In practice, the real savings in time from acquiring a range of products from a software system's library data base may lie more in the fact that the adversary has dramatically reduced the risk of taking a "wrong" design approach, than in the physical acquisition of the products themselves. No matter how long it takes the adversary to build the system once he has the pieces, he has a reasonable assurance that the pieces do fit together and will eventually provide him with an operational capability he desires. He may have substantially reduced the possibility of pouring large amounts of resources into a project that eventually will have to be scrapped or that will drag out so long as to make the product completely obsolete or functionally unnecessary by the time it is completed. This seems to be one of the main reasons the Soviets decided to functionally duplicate the IBM S/360 and S/370 computers[8].

7.3. Acquiring Military Capability

In this section we explore two additional, but distinctive, forms of technology transfer that might be employed by an adversary to improve his militray software systems capability. The first case examines the possibility of acquiring a military capability through the exploitation of commercial software. The second case uses the ADEPT-50 project as a case study to explore ways an adversary could penetrate a project to acquire software technology and know-how.

7.3.1. Acquiring Military Software Through Commercial Software

For the purposes of this example, several candidate military systems were considered, and a tactical command, control, and communications (C3) system was chosen as an example to explore in more detail.

The basic functions (Figure 7.3) of the tactical C3 system are to support the field commander and his staff in the acquisition and organization of essential tactical data, in storing, retrieving, manipulating, and displaying it to support the commander and his staff in making decisions. This support

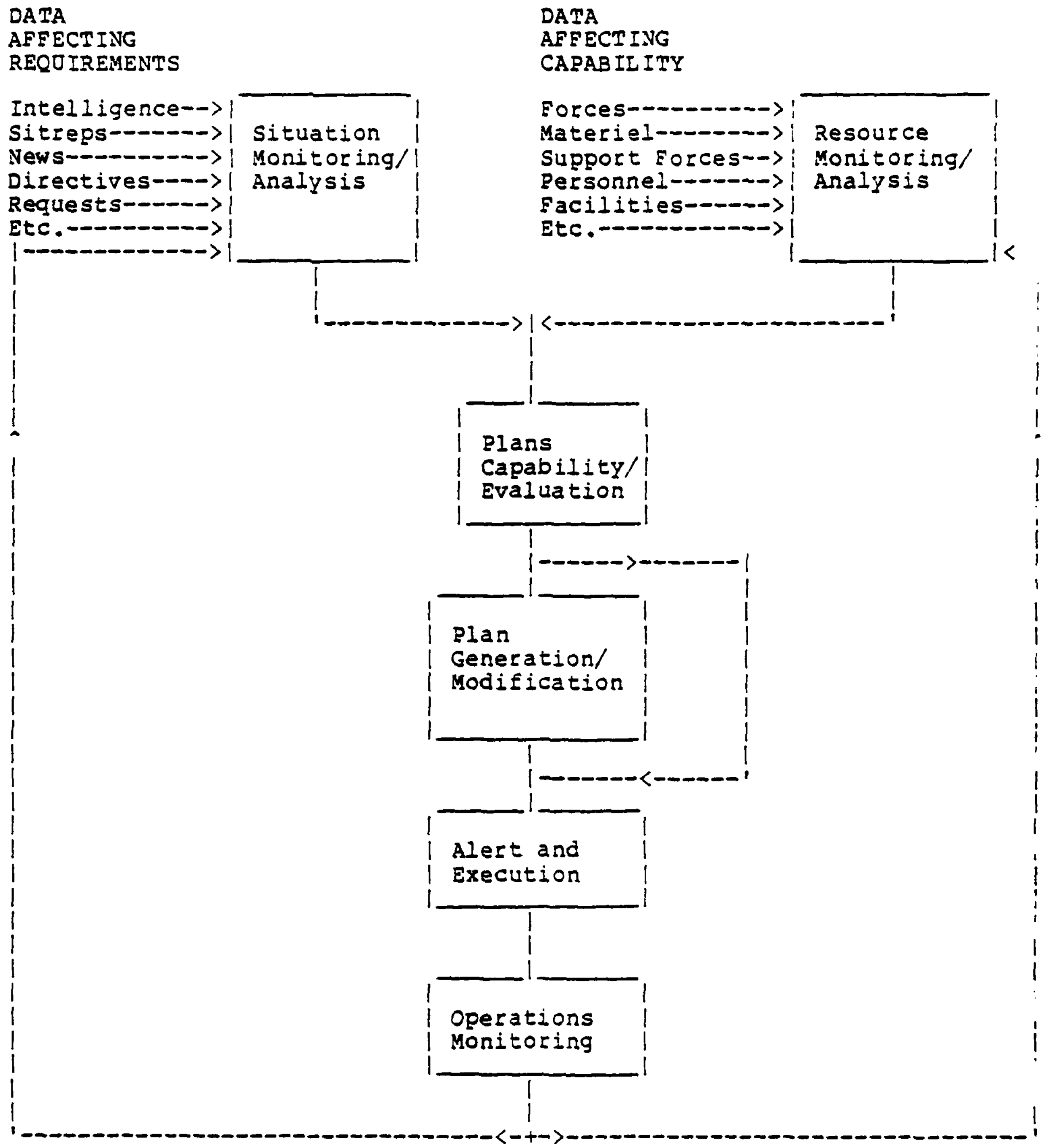

DATA AFFECTING REQUIREMENTS
Intelligence
Sitreps
News
Directives
Requests
Etc.
Situation Monitoring/ Analysis
DATA AFFECTING CAPABILITY
Forces
Materiel
Support Forces
Personnel
Facilities
Etc.
Resource Monitoring/ Analysis
Plans Capability/ Evaluation
Plan Generation/ Modification
Alert and Execution
Operations Monitoring

Figure 7.3. Basic C-cubed Functions

requires an on-line, real-time information processing system designed to facilitate effective management of field resources, particularly during emergency situations.

These functions are basic to the discharge of command responsibilities. Each of these functions is supported by one or more specific operational capabilities; i.e., by a set of interrelated computer programs designed to provide a console operator with command and status information relative to operational problems. The total set of operational capabilities provides for the retrieval of information relative to the current status of military resources, including forces, material, facilities, personnel, medical items and communications-electronics items. They also provide for the retrieval of plans information based on operator input of descriptive qualifiers about a specific military situation. Plan requirements can be compared with current status data to evaluate the feasbility of implementing a particular plan. In the event that a plan is not feasible, it may be modified through the utilization of pre-stored or operator entered planning factors. These planning factors are also used as a basis for generating plans in the event that no suitable plan exists. Subsequent to the implementation of a plan, operational reports are compared with planned events to monitor the progress of the operation. Potential problem areas can be detected and resolved quickly.

The basic software functions that must be executed in the system are:

1. Communications Processing

   The C3 system must be coupled through a communications network to higher and lower echelons of command, to other elements of the command structure, such as intelligence, logistics, etc.

2. Data Base Management

   The organization, storage, and retrieval of information in the system requires a data base management capability.

3. Terminal Control and Displays

   The system must be on-line and interactive to be effective. This requires the use of terminals for query entry and control, printers for hard copy, and the use of displays for dynamic presentation of data and graphics.

4. Special Applications Processing

   There will be a need for special functions to support command decision making. These can range from basic statistical packages to more sophisticated decision making tools.

The first stage of the effort would be to develop and test a functionally equivalent C3 system by integrating commercially available hardware and software components. The basic hardware elements needed are:

1. A central processor and main memory;

2. Mass storage peripherals (high-speed, high-density disks);

3. Alphanumeric display terminals with keyboards and a graphic display system; and

4. A communications controller.

The basic software elements are:

1. A real-time, event-driven multiprogramming system;

2. Terminal and graphics processing software;

3. A data base management system; and

4. Special applications software including analyst support and management/command decision making.

Such a system could be assembled by using a DEC PDP-11/70 data processing system*, equipped with a high-speed control and a mass storage peripheral for information storage. Alphanumeric displays and keyboard entry of data and commands could be supported by DEC VT52 terminals. Color graphics capabilities could be added by utilizing the AYDIN display system which has hardware and software designed to interface with the PDP-11/70. The operating system could be the DEC RSX-11M system which is a real-time, event-driven system that supports a variety of I/O devices, including alphanumeric displays, communications devices, etc. If the communications processing load becomes too heavy for

---

* At least two SM minicomputer models are currently being built in the Warsaw Pact countries which are based on the DEC PDP-11 architecture. We do not know to what extent these machines are compatible with those of DEC, nor whether or not the Soviets or Eastern Europeans have acquired any of the products described in this section.

for the PDP-11/70, then a DEC PDP-11/04 could be added to act as a front-end processor.

For the data base management capability a system such as DEC's Datatrieve 11 could initially be utilized. This is an interactive query, report, and data maintenance system which provides facilities for data retrieval, formatting, report generation, etc. It runs under RSX-11M and includes the RMS-11K record management services software. The one area for which commercial software is not generally available is that required to support command decision making, although a great deal of R&D is being done in the areas of self-adapting systems and computer-based information decision and forecasting systems, and software is available[1].

Once the commercial software/hardware capability is workable, then steps can be taken to make the system suitable for field military use. This can be done by van-mounting the hardware to protect it, by ruggedizing the hardware, or by acquiring military versions of the commercial hardware. For example, Norden builds a Mil-spec version of the PDP-11/70.

7.3.2. Technology Transfer Mechanisms and the ADEPT-50 System

In this section we perform a gedanken experiment on a real system development project of the late 1960's. The current Soviet systems capability can be likened to the U.S. capability of this time frame. The experiment examines the most effective technical transfer mechanisms employed on that project which, by analogy, might be employed by Soviet systems software people in current technology transfer efforts.

7.3.2.1. Project Overview

ADEPT-50 [11] was a three year project to build a general purpose timesharing system for potential military use on medium scale commercial computers. An IBM 360/50 was selected as the base hardware. The system requirements called for three classes of software, which had to be fully integrated with one another:

1. The ADEPT timesharing operating system;
2. An interactive data base management system, TDMS; and
3. A set of software development tools oriented around a JOVIAL compiler.

None of the software items existed, though experience with similar software existed in the project. Therefore, except the IBM assembler and some IBM loaders, all system components and all tools to develop the system had to be built from scratch.

The project began with four experienced designers working on the operating system (OS) and, at its peak, grew to include some dozen software people. The service functions of the OS were designed first and became the interfacing specifications for both the TDMS and tools groups. A first kernel OS was servicing these groups at the end of six months. New releases appeared every three months, slowing to every six months in the last year. Such releases were incrementally more capable systems. Software library maintenance tools, debug tools and sysgen (system generation) tools were completed during the first six months. The sophisticated JOVIAL tools were at the state of the art in design of incremental on-line compilation throughout the full project life. The TDMS system was the most advanced DBMS then conceived employing fully inverted files, English query, flexible report generation, and batch update. The OS advanced the technology of secure timesharing and timesharing for large user programs on moderate hardware. The whole system was completed successfully and installed in a number of operational DoD test beds. It survived many years until passed by modern systems. The TDMS and many tools survive today in operational use, and the ADEPT security approach began a serious effort on, and contributed to, today's multilevel secure systems.

7.3.2.2. Experimental Results

The gedanken experiment provided insight into the nature of the interrelations between the ADEPT-50 technologies and the transfer mechanisms. Much of this insight is captured and explained elsewhere in this report. It supported the choice of most of the recommended list items of Section 8. In this section we summarize the results of the experiment and the tie-ins to the list items.

a) Operational Capability

Access to the initial ADEPT-50 operating system (OS) in the first six months provided a quick operational capability for the project technical staff. This usage allowed staff personnel to become fully acquainted with the OS behavior, performance, command language, and capabilities. This, in effect, provided them with a working exposure to the system requirements and functional specifications. The importance of this experience cannot be underestimated, because it had considerable impact on many of the tasks in system development, e.g., online documentation, coding, and testing.

b) Source Code and Specifications

Of course, these system development tasks also required access to system source code and specifications contained in the system software library data base. This added material significantly increased technical understanding, and was necessary for the staff to extend the operational capability by system modification as noted below.

It seemed to us that adversary programming groups could, and have, benefitted in similar ways from access to operational capabilities and parts of the software library data base. We believe that what has been described in the preceding two paragraphes were major inputs in the efforts by the Warsaw Pact countries to acquire the IBM S/360 operating systems for their Unified System (ES) family, and may well be an important aspect of other software acquisitions by these countries. A goal of export control should be to try and limit access to the most revealing parts of the software library data base, as we try to do with the list item described in Section 8.2.2. Furthermore, in Section 8.3, we suggest that access be limited to the least revealing parts of this data base, i.e., to object code and basic users manuals, and the use of object code that is somehow technically configured to limit portability.

c) Software Development Tools

If access to the ADEPT-50 technical data base made extentions to the system feasible, access to software development tools made such extensions practical. These tools are described in Sections 2.3 and 5.5, and allow controlled manipulation and management of the source code. Since they themselves are software, they are subject to all the limitations and foibles involved in building any large-scale software product. They are indispensible in any large-scale software development project, are key to software export control, and are addressed by an important list item described in Section 8.2.3.

d) People

The ADEPT-50 project began with a small, experienced staff and grew to include more junior technical people as the project matured. These junior staff members were apprentices to the senior designers, who acted as consultants and mentors to the junior staff. They were the "active" transfer agents, explaining why the essential features of the design were selected and how they were implemented, assisting the new members in finding their way around the extensive and growing software library data base, and teaching the junior staff how to make the best use of the software development tools. Coupled with the other aids noted above, the experts greatly improved the efficiency of the team. The active mechanisms used here (and others) have been described in Section 6, and have been partitioned into four categories by their effectiveness and controllability (Section 6.5). The recommendations for restricted mechanisms form an important part of the definitions and potential implementations of all our recommended list items (Sections 8.2.1-8.2.8).

e) System Modification

We believe that software operations and maintenance (O&M) are really software redesign and implementation. Therefore, the technical ability to maintain and enhance software is technology that can be used to build software. The ADEPT-50 experience is typical of the development of many large software systems, and supports this point. The new ADEPT-50 releases which were issued were essentially new software products, even though they were composed largely of the same modules as existed in the earlier releases, plus new functional additions and changes to some of the older modules. The importance of this maintenance technology is such that we felt the need to include it as a distinct list item, described in Section 8.2.4.

f) Conclusion

It is relevant to our present analysis to note that, within a year, the junior staff were sufficiently skilled, trained, and experienced to support the configuration management and production of new releases. After another year or two, they were leading software development projects of their own. By analogy, a Soviet team might be able to repeat the learning experience of the ADEPT-50 junior staff by initially taking responsibility for maintenance of a commercially available U.S. software product. Mastering the maintenance tasks would allow the staff to begin to make incremental and selective modifications or to adapt the software for other, including military, applications. It appears that some of the Warsaw Pact countries have been doing this. Furthermore, use of the more active transfer mechanisms would greatly facilitate such transfers, and would contribute further know-how that could better enable adversaries to build new systems for military purposes.

7.4. Software Development Trends

In order to combat the proliferation of CPU architectures and to shorten the development cycle and to reduce costs, the military services are moving towards the use of so-called commercial test beds. The idea is to develop and field prototype systems for feasibility demonstration using commercial hardware and as much "off-the-shelf" software as possible. As a matter of fact, the Army is using an approach not unlike that postulated in Section 7.3.1. for its Beta system.

Furthermore, the trend is to create standardized software development systems to be used by both military and contractor personnel to develop and maintain military software systems. As an example of this, the Army is proposing to create 11 PDSS (Post Development Software Support) centers for the development and maintenance of all Army battlefield automation software. These centers will be equipped with common hardware and support software and tools. The use of Ada as the standard programming language, plus the rigid enforcement of software standards and practices, will be an integral part of the process. These trends, plus the standardization of military computer architecture families, will make it easier in the future for a determined adversary to acquire software systems capability if adequate controls are not exercised.

8. Recommendations

8.1. Introduction

This section presents our recommendations for list items on the militarily critical technologies list and a number of other recommendations concerning product form and further study. The posture of this group has been to consider software technology and its transfer as a process. Therefore, we feel that controlling related technology transfer mechanisms is an integral part of the definition and control of software technologies. For easy reference, we have reproduced the four categories of transfer mechanisms from Section 6.5.2.

Categories of Software Technology Transfer Mechanisms

The following four categories of transfer mechanisms apply regardless of the source of the transfer (academic, governmental, industrial):

Category I

Effective mechanisms for which adequate controls may be possible.

Apprenticeships:	Team Membership License with Extensive Teaching
Joint Ventures	
Product Shipment:	Development Data Base
Product Support Assistance:	Modification and Enhancement Training Maintenance Training
Consulting	

Category II

We feel that these mechanisms are less effective and more difficult to control than those in Category I, but that control may be possible.

Professional Contacts:	Extended Visits (without apprenticeships) User Group Participation
Product Shipment:	Source Code and Detailed Documentation
Product Support Assistance:	Installation Training
Academic and Other Courses:	State-of-the-art training (including courses), not widely available
Contract Negotiations:	Detailed negotiations and proposals
Usage	

Category III

While control of the following mechanisms may be possible, it is doubtful if such controls would be desirable or worth the effort of imposing them.

Professional Contacts:	Control of visiting adversary nationals' movements in those parts of host organization's facilities that are accessible to all members of that organization (e.g., restricted access to computer centers at commercial or non-profit organizations). Movement of U.S. citizens abroad "Licensing" people who deal with adversary countries in technical matters

Category IV

We feel that control of the following mechanisms would be highly undesirable either because they only involve weak technology transfers or because controlling them would impose severe constraints on the software industry in the United States.

Written Material:	Widely Available Literature (not classified or proprietary)
	Correspondence
Product Support Assistance:	User Training for Purchased Products
Product Shipment:	Object Code and Users Manuals
Professional Contacts:	Short Term (less than 2 weeks) Contacts and Visits
	Short Term Visits to Trade Shows, Exhibits, and Conferences
	Work that is Primarily of a Theoretical Nature
Apprenticeships:	One-on-one Thesis Supervision (but restrict exposure to facilities or group projects)
Academic Courses:	Courses that are Widely Available

8.2. Recommended Entries for the Militarily Critical Technologies List

8.2.1. Life-Cycle Management Technology

Description

The best working software is a product of a number of discrete stages with defined output and review, i.e., a social process. Together with the use of the software library data base and software development tools, they comprise what we have chosen to call "life-cycle management technology." Although numerous approaches to the life cycle for software are used, one that is modelled here is the DoD methodology.

The earliest stage is that of _concept definition_, when the overall system purpose and operation is conceived. A clear statement of objectives is required. Objectives may be derived from higher-level systems, from control of lower-level systems, from simulations, and from "war gaming" scenarios. Cost and scheduling factors also assist in the concept definition.

The _requirements and specifications_ stages begin to structure what the system must do to satisfy its objectives. Again, simulation can be employed. Techniques of structured requirements are usefully employed to follow the flow of system operational control and its needed data and computational requirements. Once developed, these requirements and their specifications must be written in well-formed, unambiguous notation. A number of such languages now exist and are used in preparing mathematically precise system specifications of what the system must do, i.e., its service specifications.

System design is the stage that defines how the system works, i.e., how the system implements the service specifications. A design may be written in at least one of a number of notations: flow diagrams and data diagrams, state machine tables or specification languages, English, structured English, or even a programming Higher Order Language (HOL) of the coding variety. Each approach carries with it advantages and disadvantages and a considerable technical methodology. All approaches use a form of modular design which defines the input, output, and functional behavior of each module. With the definition of these modules and their interfaces to other modules, system hardware, and human components, a software architecture is developed. More modern methods go further in describing the types of parameters, and their "visibility" in scope to other modules. Side effects and environmental considerations for each module may also be specified.

Coding proceeds directly from the design stage. First the individual modules are coded, then the associated modules until a chain of integrated modules is built up which performs one or more of the service specifications. The design is often implemented in an HOL such as FORTRAN, COBOL, JOVIAL, PASCAL, etc.

These module chains, called "builds," relate directly to the requirements specification and form the basic unit of testing of the system. Modern testing methods employ "threads" or "builds" testing which checks the correct operation of a thread (i.e., a logical, ordered subset of the whole) of functions which satisfy one system requirement. There is considerable technology required for system testing. Test plans must be developed to lay out a strategy of tests to be performed in sequence by a number of systems people working in parallel. Test conditions are set up, parameters to drive modules are created, and results are captured and analyzed against specifications and requirements. Errors that are found must be logged and engineering changes generated and controlled for correcting such errors. The module inventory grows and changes with each engineering change, and a system of controls tracks software releases and the errors outstanding against them. These tests proceed thread by thread until all requirements are demonstrated. Threads are then merged into a complete **integrated** system, which is tested for correctness and performance. Lastly, the system is tested at the user's installation. This may be the first time that all of the system elements work together and use real ("live") data.

A number of management techniques are used at various points in the life-cycle. Standard management activities, e.g., preparing work breakdowns, cost estimates, schedules, and manpower loading statements, have been adapted to the peculiarities of the software industry. Some of the know-how which has been acquired through difficult learning experiences includes understanding and anticipating the rigor needed for a large software project, handling detailed internal management and technical documents, incorporating a number of defined events and milestones for management review at various levels, using modelling and control systems, and building project management on the basis of hierarchies of individuals who have different levels of experience and responsibility. Much of this experience cannot be acquired through open, passive sources.

List Entry: Software Life-cycle Management Technology

Integrated technical information and know-how related to the understanding and utilization of life-cycle management methods for the development of large software systems. Integrated technical data and know-how include the aggregate of methods, procedures, manuals, standards, events and milestones, work-breakdown modeling and analysis techniques of resources (e.g., cost, schedule, labor, equipment) embodying U.S. management experience. Large software systems are those involving over 15,000 HOL source statements, or four or more person years of labor before the initial delivery of the system.

Military Utility: High

Foreign Availability: See summary in Section 3.4.2.

Adversary Capability: See summary in Section 3.4.1.

Recommendations for Controlled Transfer Mechanisms: Mechanisms in categories I and II listed above.

8.2.2. Software Library Data Base

Description

Unlike finished goods in other technical manufacturing, software has no single physical form. In the early life cycle stages it is English functional descriptions. In design stages it is more akin to formal mathematics or logical information flow specifications. In the coding stage it is in the form of "source" text in a HOL. This text is translated by software tools into "object" binary form for direct execution on a given computer. Application programs in HOL source code form may be translated into many different object code forms for different computers, or into different object code forms on the same computer for various configurations of interfacing software and peripheral hardware. And for each form there must be accompanying documentation to describe the software operation and differences.

Finally, through the O&M process, the source programs are changed, repaired, and improved in function and performance to produce an assortment of new versions of essentially the same "product." The aggregate of these software items typically constitute millions of lines of text in various forms. If any one of these items is incorrectly formulated or maintained, incorrect operation can result.

In order to maintain these items over the course of the software life cycle, a software library data base is used. The data base is created incrementally and is a "living" document, best maintained on-line by a sophisticated set of tools. The structure of the data base depends on the conventions of the languages and notations employed in the various stages of development. These conventions must permit both human and machine access to the text.

The data base contains multiple directories of the objects in the data base. All directories originate from a master directory, which is often organized along system or component lines so that releases of modules are placed with other modules of the same thread or function. Subdirectories often follow the organizational structure of the development project, with each programmer having his or her private files. These files are periodically released to the software librarian to include in the master directory. The master directory is further organized by text forms for each development stage; it then resembles a multi-dimensional matrix of functions, forms, and people. This structure is key to the retrieval and to the automation of the generation of system products: software and documentation. Even the naming of objects becomes a crucial technology as much of the structure is embedded in the names. They reflect a path through the directories, and they encode the form of the text and version number of interest (e.g., JONES.PASCAL.3). They also provide a uniform key upon which all of the related software tools can operate.

Access to the library is strictly controlled by the librarian and operating procedures for obvious reasons of safety and protection, but also for less obvious reasons of error control, cost control, status reporting, and project communication. Private files are strictly controlled by each owner. Backup procedures are of highest priority and are handled both by the librarian and by individual programmers. Backup becomes more crucial and more costly as the library grows.

The techniques outlined above are collectively known as "configuration management."

List Entry: Software Library Data Base

A large software product, which is the aggregated set of the final textual forms it takes at each stage of the life-cycle development process; and the data base and configuration management techniques associated with these forms. These forms are English descriptions, specification and design expressed in special languages, HOL source code, and machine-executable binary object code. They are entered into a software library data base and are controlled through this data base and configuration management techniques. Large software systems are those involving over 15,000 HOL source statements, or four or more person years of labor before the initial delivery of the system.

Military Utility: High

Recommendations for Controlled Transfer Mechanisms: The mechanisms in Categories I and II listed above.

8.2.3. Software Development Tools

Description

Software development tools vary widely and encompass almost the full range of activities associated with software development. It has taken the U.S. a long time to realize the need for the systematic software development practices which are "enforced" by the use of these tools. A considerable intellectual and financial investment has been made in them and in the technology that employs them correctly. For the purposes of this list item, only integrated sets of tools should be considered.

A novel feature of the DoD language Ada is that it has been designed from the start with the concept of a tools environment in mind. The building of the "Ada environment" will involve a very substantial software life-cycle management effort; our controls extend to this experience and the environment, but not to the language itself. The Ada environment is an example of an integrated set of tools.

Software development tools fall into a number of generic categories. Library maintenance tools are used to create directories and files, for storage and retrieval of text, and for other data base operations. Composition tools are used to enter, edit, display, and/or print software text. Translation tools include compilers, interpreters, assemblers, macro libraries, pre-processors, and post-processors. Test and validation tools confirm that the system is working correctly. Finally, project management tools deal with all the administrative aspects of the system being built.

List Entry: Software Development Tools

Array of technical information and know-how consisting of the integrated set of software development tools used in the development of large software products, including tools for program library maintenance, program composition, translation, test and validation, and program project management. "Integrated" means the use of project management or library maintenance tools; or the use of either of these in conjuction with any of the other above-mentioned tools. Furthermore, these tools must be designed to interface with each other's internal data structures, command language, and internal (process) communications between modules.

Military Utility: High

Foreign Availability: Moderate.See Section 3.4.2(c).

Adversary Capability: Limited. See Section 3.4.1(f).

Recommendations for Controlled Transfer Mechanisms: The mechanisms in Categories I and II, given above. Furthermore, products in any form, including object code, should be controlled.

8.2.4. Maintenance of Large Software Products

Description

The strategic value in software is not in a "secret formula," but that it "works" and can be relied upon to continue to operate as expected. Therefore, the system enters operation and maintenance (O&M) after final testing. The O&M stage might be considered the end, because the system is completed and in operational use. But large programs are quite complex and involve many interfacing interrelated "clockwork" mechanisms. They are very fragile in the sense of needing continuing support and maintenance to keep them current and operational. Repairs are needed to correct errors, upgrades to improve performance, changes to accommodate hardware configuration changes or new performance requirements. New functional capabilities unforeseen or unclearly outlined in the original specifications may need to be added. Such modification of working software is the rule of industry and is reflected in the model, version, or release numbers associated with all software products.

Such repair is really redesign and requires a return to earlier life cycle stages. Thus, O&M must make extensive use of the documentation of the program contained in the software library data base. Knowledge required to carry out O&M involves understanding the software architecture, detailed design, the various specification and programming languages in which the software is written, the computer on which the software operates, the complement of equipment in the system configuration, the applications environment, the types and ranges of expected input and output variables, how to operate test tools, and the proper use of configuration management tools to keep the software current. O&M is a major problem stage for software, because it is entered years after the concept stage and when few of the original designers are available to perform the changes. Overall, O&M costs about twice the total of all other stages combined. While O&M follows the other life-cycle phases sequentially, it really must be considered an entirely separate stage. Few, if any, of the personnel who built the software continue to support it and O&M is carried out for an extended period of time after operation begins.

List Entry: Maintenance of Large Software Products

Integrated technical information and know-how related to the understanding and utilization of life-cycle management methods for the maintenance, i.e., repair, adaptation, extension, or modification, of large existing software systems, or large software systems developed under life-cycle management methods. Integrated technical data and know-how include the aggregate of methods, procedures, manuals, standards, events and milestones, work-breakdown modeling and analysis techniques of resources (e.g., cost, schedule, labor, equipment) embodying U.S. management experience. Large software systems are those involving over 15,000 HOL source statements, or four or more person years of labor before the initial delivery of the system.

Military Utility: High

Adversary Capability: Limited

Recommendations for Controlled Transfer Mechanisms: The mechanisms in Categories I and II, given above.

8.2.5. Formal Methods and Tools for Developing Trusted Software

Description

An emerging technology of great promise is that of formal methods to produce software and the tools to enforce the use of these methods. This technology utilizes theoretical results from mathematical logic (particularly the predicate calculus), packaged using heuristic techniques from the artificial intelligence community, to develop software systems which verify the correctness of software specifications. From the formal results have also come methodologies to discipline the programming process, so that resulting products are more amenable to proof of correctness, and sets of languages and verification tools (e.g., specification languages and their processors, theorem provers, specialized configuration control tools) to enforce the discipline that theory recommends.

Formal methods enforce a top-down approach to the production of software, starting from formal specifications. Tools have been developed to express, enforce, and prove the correctness of these specifications and to refine them to more detailed specifications which, in turn, can be proved correct and further refined. Iteration of this procedure produces a hierarchy of proved specifications, finally resulting in a proved specification sufficiently detailed to permit direct coding. This software is then considered "trusted."

List Entry: Formal Methods and Tools for Developing Trusted Software

Technical information and know-how embodied in arrays of formal methods and tools which permit the development of trusted software, proved to satisfy its specifications. Formal methods include specification languages and their processors, theorem provers and specialized configuration control tools. The tools permit expression, proof, and enforcement of correctness of software to its specifications at several levels of detail.

Military Utility: High

Recommendations for Controlled Transfer Mechanisms: The mechanisms in Categories I and II, given above. Furthermore, products in any form, including object code, should be controlled.

8.2.6. Secure Software

Description

The formal methods discussed in Section 8.2.5 have permitted the development of an emerging technology of secure software for large, complex software systems. This software permits computer systems which are "trusted" to instantiate predetermined security policies. In particular, this software may permit multi-level secure operating systems or secure data base management systems which guarantee that users not have access to information for which they do not have appropriate access permissions. Much of this software may be classified and thus fall under the control umbrella of the Munitions Act, but the development of secure systems is also necessary for such applications as electronic funds transfer.

List Entry: Secure Software

Software (e.g., multilevel secure operating systems, communications systems, special controllers, and secure data base management systems) that is produced using formal methods and tools so that it is trusted to adhere to predetermined security policies. Secure software embodies technical information and know-how on systems architecture which is resistant to penetration.

Military Utility: High

Recommendations for Controlled Transfer Mechanisms: The mechanisms in Categories I and II, given above. Furthermore, products in any form, including object code, should be controlled.

8.2.7. Large Self-adapting Software Systems

Description

Self-adapting software systems are able to make significant changes in their internal processing logic in response to user commands or based on the demands which have been placed on the system in the past. Such systems usually incorporate significant artificial intelligence technology. Examples are the MACSYMA system which reorganizes internal data representations and secondary access paths based on usage statistics; the "Programming by Example" system which automatically generates significant programs from very succinct and natural user descriptions; and natural language user interfaces, especially those which can "learn" new vocabulary.

Self-adapting software systems are important because they create significant operational capabilities across a diverse set of applications. The breadth of the applicability makes these systems fundamentally superior to more rigid systems which are only useful with complete documentation and maintenance tools.

List Entry: Large Self-Adapting Software Systems

Any computer system which automatically or semi-automatically makes significant revisions to its internal software processing logic and data in response to user commands or based on the history of demands placed upon it.

Military Utility: Very great potential utility in command and control and intelligence operations.

Foreign Availability: There are few, if any, significant systems of this type outside the United States. The European perception that computer hardware is expensive and must be parsimoniously utilized has retarded their development of systems of this type. Japan has been exploring such systems but does not yet have (known) significant results, except in the robotics field.

Adversary Capabilities: There is no known significant capability in this area in any adversary country.

Recommendations for Controlled Transfer Mechanisms: The mechanisms in Categories I and II, given above. Furthermore, products in any form, including object code, should be controlled.

8.2.8. Commercial Software Integral To Critical Military Systems

Description

A substantial amount of military software in current defense use is unclassified or is available as commercial software products. Military systems make use of general purpose software including operating systems, communications front-ends, a wide variety of software utilities, and various applications packages.

Such software is of concern because it can potentially be acquired and used to expose flaws in critical U.S. military systems which may then be exploited in some fashion. Penetration studies, for example, have shown that significant information about flaws can be derived from testing the operational characteristics of a given program.

To meet this threat of "jamming" U.S. systems, the software subgroup feels that explicit commercial products which are in use in critical military systems should not be transferred to adversary nations. A classified list of such products should be compiled by DoD and used in the export administration process. These controls should not extend to commercial software products which perhaps provide similar operational capabilities to those explicitly in use.

<u>List Entry</u>: Commercial Software Integral to Critical Military Systems

Commercial software products (e.g., operating systems, file and data management sysytems, communications front-ends) which are integral to existing critical U.S. military capability or could be reverse-engineered to expose flaws in U.S. operational military systems.

THE DEPARTMENT OF DEFENSE SHOULD ADD A CLASSIFIED LIST OF EXPLICITLY NAMED PRODUCTS TO SUPPLEMENT THIS ITEM.

<u>Recommendations for Controlled Transfer Mechanisms</u>: The mechanisms in Categories I and II, given above. Furthermore, products in any form, including object code, should be controlled.

8.3. Other Recommendations

8.3.1. Recommendations About Product Form

In addition to identifying technologies which should be included on the militarily critical technologies list, we feel that attention should be given to the form in which any software product is exported.

a) We view object code as a fairly weak form of encryption, since partial or complete decoding algorithms are available for commonly used languages and machines. Nevertheless, the time and talent that must be invested in decompiling object code for large software systems can be very substantial. Certainly the possession of well-documented source code can be very helpful to an effort to functionally duplicate a product, or to an effort to diffuse and maintain the original product. We believe that the export of object code, or ROM or other "hardware" forms, is to be preferred over the export of source code. Another way to strengthen object-code-only export is through license control simplification for software embedded as part of a "turnkey" product with U.S. maintenance **only**. Object code, or software in a hardware form, would then be a part of the total product, the internal structure of which remains unknown and unknowable to the end user; e.g., a form of commercial "need-to-know."

b) Computer manufacturers and software houses are likely to find increasing economic reasons to support hardware and software mechanisms to aid in controlling the diffusion of software. This trend may do much to help solve some of the problems of the export control of software products. Greater use of turnkey products is one method. Cost effective technical means to restrict portability need to be found and made widely available. We urge the government to support efforts to develop such mechanisms.

c) The transfer of any software product should be uncontrolled if the product does not provide a direct military operational capability or explicitly fall into a small number of categories (e.g., software tools or self-adapting systems), provided that it is transferred in the form of object code with only users mannuals and passive maintenance service, and provided that there be reasonable technical safeguards that the system not be portable.

8.3.2. Recommendations for Further Study

a) Other Software Items for the Militarily Critical Technologies List

Although we feel we have covered the most important software technologies, no claim is made that the eight recommended List Entries cover this field exhaustively. Other general, emerging software technologies, analogous to the self-adapting software entry (8.2.7), may have to be listed. One such possibility is robotics software. Time and expertise constraints prevented the software group from making an exhaustive study.

b) Technical Measures to Reduce Software Portability

The software subgroup strongly recommends that research directed towards the development of technical measures to reduce software portability (Section 6.5.2) be funded by the Federal government. One possibility involves hardware "signatures" to bind software to specific hardware configurations. Previous work in this area has been cost-limited, but present decling hardware costs and increasing investments in software should make such measures economically viable.

c) Construction of a Decision Tree

It may be possible to use the analyses of this report to construct an explicit decision tree for evaluating software-related export requests.

d) Further Thought Experiments

We consider analyses such as those of Section 7 to be very valuable in providing a deeper understanding of the nature of technology transfer. From a broad perspective, deepening our perception and insights about software technology transfer can enhance the search for effective control methods and the delineation of what should be controlled. From a more narrow perspective, such analyses help to illuminate such issues as how militarily significant systems may be constructed from commercial software or how U.S. systems may be penetrated and "jammed." Hence, we recommend that anaylses of this nature be expanded and continued.

e) Increased Awareness

It is necessary to increase the level of awareness of the general public, the computing community, and those involved in the export control process about the concerns raised in this report. Technology transfers have taken place as a result of ignorance of what it is about software that is of concern. Any effective control of software technology transfer will have to involve groups that had not considered their activities involving adversary countries to be subject to prior review or voluntary restraint.

f) Implementation of Transfer Mechanism Restrictions

The controls on transfer mechanisms recommended by this subgroup go considerably further than the license/approval system now in use at the Commerce Department. Consequently, serious consideration will have to be given on how to implement these recommendations.

g) The Burden on Working DoD Computer Scientists

The evaluation of export requests often appears to disrupt the work of DoD computer specialists, who may find such requests to be an unmanageable burden on top of a full schedule of unrelated duties. These people may also lack knowledge of adversary capabilities, foreign availability, and the process of software technology transfer. The net result is a disruption of DoD computer-related work, weakly informed decisions, and delays suffered by potential exporters. The Department of Defense needs a few suitably knowledgeable people, with good communications to DoD and industry, for whom the evaluation of potentially threatening computer technology transfers is a major task.

h) Coverage of Foreign and Adversary Software Capabilities

Coverage of these areas by U.S. government organizations is weak and needs to be strengthened considerably.

References Cited

1. Andriole, S.J. "Another Side to C3." Signal, pp. 15-22.

2. Auerbach, Isaac L. "Computing in China, 1979: An Update." Computer 12, Number 11 (November, 1979), pp. 52-60.

3. Brooks, Frederick P. Jr. The Mythical Man-month: Essays on Software Engineering. Reading, Massachusetts, 1975.

4. Computer Network Critical Technology Expert Group (F.R. Spitznogle, Chairman). Computer networks: An assesment of the critical technologies and recommendations for the controls on the exports of such technologies. Final report, prepared for the U.S. Department of Defense, April 30, 1979.

5. Data and Analysis Center for Software. The DACS Glossary: A Bibliography of Software Engineering Terms. October, 1979, (Ordering Number: Glos-1).

6. Garner, Harvey L. "Computing in China, 1978." Computer 12, Number 3 (March, 1979), pp. 81-95.

7. Gold, C. L., Goodman, S. E., Walker, B. G. "Software: Recommendations for an Export Control Policy." Communications of the ACM 23, Number 4 (April, 1980), pp. 199-207.

8. Goodman, S. E. "Software in the Soviet Union: Progress and Problems." Advances in Computers 18 (1979), pp. 231-287.

9. "Hungarians Working for Western Software Firm." Heti Vilaggazdasag, January 12, 1980, pp. 30-31.

10. Kirchner, J, and Rosenburg, M. "Belgian Charged with Bribery." Computerworld, June 16, 1980, p. 1,4.

11. Linde, R.R., Weissman, C., Fox, F.E. "The Adept-50 Time-Sharing System." FJCC, 1969.

12. McLean, John. USAF Critical Technology: Computers, Volume V (Draft Report, June, 1980).